(12)

TR-311

# STUDY OF COMPUTATIONAL STRUCTURES FOR MULTIOBJECT TRACKING ALGORITHMS

FINAL REPORT



ALPHATECH, INC.
3 NEW ENGLAND EXECUTIVE PARK
BURLINGTON, MA 01803
617-273-3388

(12)

TR-311

# STUDY OF COMPUTATIONAL STRUCTURES FOR MULTIOBJECT TRACKING ALGORITHMS

FINAL REPORT

By

T.G. Allen
T. Kurien
R.B. Washburn

December 1986

Submitted to:

Office of Naval Research
800 N. Quincy Street
Arlington, VA 22217

Contract No. N00014-84-C-0378



ALPHATECH, Inc.
2 Burlington Executive Center
111 Middlesex Turnpike
Burlington, MA 01803
(617) 273-3388

SECURITY CLASSIFICATION OF THIS PAGE AD-A175 247

# REPORT DOCUMENTATION PAGE

| | |
|---|---|
| 1a. REPORT SECURITY CLASSIFICATION<br>UNCLASSIFIED | 1b. RESTRICTIVE MARKINGS |
| 2a. SECURITY CLASSIFICATION AUTHORITY | 3. DISTRIBUTION / AVAILABILITY OF REPORT<br>Approved for public release, distribution unlimited |
| 2b. DECLASSIFICATION / DOWNGRADING SCHEDULE | |
| 4. PERFORMING ORGANIZATION REPORT NUMBER(S)<br>TR-311 | 5. MONITORING ORGANIZATION REPORT NUMBER(S) |
| 6a. NAME OF PERFORMING ORGANIZATION<br>ALPHATECH, Inc.<br>6b. OFFICE SYMBOL (If applicable) | 7a. NAME OF MONITORING ORGANIZATION<br>Office of Naval Research |
| 6c. ADDRESS (City, State, and ZIP Code)<br>2 Burlington Executive Center<br>111 Middlesex Turnpike<br>Burlington, MA 01803 | 7b. ADDRESS (City, State, and ZIP Code)<br>800 N. Quincy Street<br>Arlington, VA 22217 |
| 8a. NAME OF FUNDING / SPONSORING ORGANIZATION<br>Office of Naval Research<br>8b. OFFICE SYMBOL (If applicable) | 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER<br>N00014-84-C-0378 |
| 8c. ADDRESS (City, State, and ZIP Code)<br>800 N. Quincy Street<br>Arlington, VA 22217 | 10. SOURCE OF FUNDING NUMBERS<br>PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO. |

11. TITLE (Include Security Classification)
STUDY OF COMPUTATIONAL STRUCTURES FOR MULTIOBJECT TRACKING ALGORITHMS

12. PERSONAL AUTHOR(S)
Allen, Thomas G.; Kurien, Thomas; Washburn, Robert B. Jr.

| 13a. TYPE OF REPORT | 13b. TIME COVERED | 14. DATE OF REPORT (Year, Month, Day) | 15. PAGE COUNT |
|---|---|---|---|
| Final Report | FROM 9/18/85 TO 8/14/86 | December 1986 | 162 |

16. SUPPLEMENTARY NOTATION

| 17. COSATI CODES | | | 18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) |
|---|---|---|---|
| FIELD | GROUP | SUB-GROUP | MULTITARGET TRACKING, PARALLEL ALGORITHMS, PARALLEL COMPUTERS |
| | | | |
| | | | |

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The structure and performance of multiobject tracking algorithms are heavily constrained by their current implementation on sequential computer architectures. Near-optimal algorithms, such as the track-oriented approach under investigation here, impose prohibitive requirements on sequential processors. These requirements force one to use near-optimal algorithms only in low target density environments and to employ faster but less optimal tracking algorithms in the more demanding high density situations. Identifying, enhancing, and exploiting the parallel computational structures in the near-optimal tracking algorithms potentially allows them to be applied to a wider range of scenarios. This report describes research undertaken to determine how parallel computer architectures might be utilized to implement sophisticated modern tracking algorithms which are beyond the capability of current sequential processing hardware.

| | |
|---|---|
| 20. DISTRIBUTION / AVAILABILITY OF ABSTRACT<br>☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS | 21. ABSTRACT SECURITY CLASSIFICATION<br>UNCLASSIFIED |
| 22a. NAME OF RESPONSIBLE INDIVIDUAL | 22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL |

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

ABSTRACT

The structure and performance of multiobject tracking algorithms are heavily constrained by their current implementation on sequential computer architectures. Near-optimal algorithms, such as the track-oriented approach under investigation here, impose prohibitive requirements on sequential processors. These requirements force one to use near-optimal algorithms only in low target density environments and to employ faster but less optimal tracking algorithms in the more demanding high density situations. Identifying, enhancing, and exploiting the parallel computational structures in the near-optimal tracking algorithms potentially allows them to be applied to a wider range of scenarios. This report describes research undertaken to determine how parallel computer architectures might be utilized to implement sophisticated modern tracking algorithms which are beyond the capability of current sequential processing hardware.

CONTENTS

CONTENTS (Continued)

FIGURES

FIGURES (Continued)

TABLES

ACKNOWLEDGMENT

This research was supported by the Naval Air Systems Command under Contract No. N00014-84-C-0378 with the Office of Naval Research.

# SECTION 1

# INTRODUCTION

## 1.1 PROBLEM BACKGROUND

The task of tracking uncooperative targets is a process that has become increasingly more difficult for traditional surveillance algorithms. Hostile targets have taken every action, from stealth composition to terrain following, to make themselves less visible to both active and passive sensors. Due to the improvements in counter-tracking technologies, sensors are no longer able to provide a clear and unambiguous description of the environment. The associated tracking algorithms must, therefore, be able to extract and interpret target information from confusing data. Algorithms designed in an era of less challenging targets will not be sufficient, and so more advanced algorithms must be employed. Such mathematically involved algorithms, while providing superior tracking capability, require substantial processing resources in order to be implemented in real time. In the research reported here we examine a solution to this computational burden -- the implementation of advanced multiobject tracking algorithms on parallel computer architectures.

The identification and exploitation of parallel computational structures in multiobject tracking algorithms are crucial to their successful application in many realistic scenarios. The implementation of the optimal or near-optimal form of such algorithms imposes prohibitively severe computational requirements on current-day sequential processors. These requirements force

one to use near-optimal algorithms only in low target density environments and to employ faster but less optimal tracking algorithms in more demanding high target density situations. Exploitation of parallel algorithmic structure potentially allows one to apply near-optimal tracking algorithms in real time in high-density, low probability of detection measurement environments where present tracking methods cannot resolve measurement ambiguities and thereby fail to track adequately.

## 1.2 RESEARCH OBJECTIVES

The research described here addresses the issues mentioned above by investigating the parallelism inherent in a previously developed multiobject tracking algorithm, that of track-oriented hybrid state estimation [1] - [4], in order to determine the possible improvements in tracking capability attainable with parallel computer architectures. In contrast to the simpler methods, the track-oriented algorithm can resolve measurement ambiguities through its multiple hypothesis approach. That is, many different interpretations of the environment are retained until such time that discrepancies can be resolved. It is these multiple hypotheses that provide both the improvement in tracking performance and the increased burden on the data processing hardware that necessitate this study.

The selection of the track-oriented approach over other multiobject tracking algorithms for this investigation is justified by several factors. As mentioned previously, it is an advanced algorithm that provides near-optimal tracking performance. Additionally, it has the potential for even better performance on advanced computers than on sequential computers, since the approximations necessary for application to sequential computers may be

removed. Another factor is that, as the track-oriented algorithm is a particular implementation of optimal multiobject tracking, and not a specialized ad hoc approach, analysis of it will provide considerable insight into the form and characteristics of the optimal algorithm. These insights may then be applied to other tracking methods and implementations, especially other multiple hypothesis approaches. Finally, and perhaps most importantly, the track-oriented algorithm has a large degree of obvious parallelism (filter operations, track-measurement associations, etc.), and a substantial amount of non-obvious parallelism (multiple hypothesis functions) that are a significant portion of the computational load.

It should be emphasized that this algorithm, along with most other state-of-the-art multiobject tracking methodologies found in the literature, assume a sequential implementation in their derivation. Hence, no effort was made to exploit the parallel nature of the tracking problem. It is felt that by restructuring such algorithms the computation may be spread out across many processors, thereby increasing both the algorithm speed and the number of targets that may be handled. How this may be accomplished for the specific example of the track-oriented algorithm is the objective of this research.

## 1.3 APPROACH

The approach taken here was to investigate the computational structure of the track-oriented multiobject tracking algorithm, determine the intrinsic parallelism available either directly from the algorithm or by restructuring the computations, and to explore possible methods for implementing the algorithm on representative multiprocessor computer architectures.

As this research is an initial foray into multiprocessor applications of this specific algorithm, it is important that we start with an examination of the computational structure and extent of the algorithm itself, independent of any specific computer system. Only by ascertaining the amount of exploitable parallelism inherent to the algorithm will it be possible to make an informed analysis of possible hardware configurations. Also, it is crucial that before any specific applications are considered the algorithm be well analyzed and understood. It is for these reasons that the work here focuses on the algorithmic side of the problem more heavily than on the computer architecture side.

An algorithm generally exhibits prallelism at various levels of granularity at which operations can be defined, from the instruction level up to the program level. We have chosen to investigate the parallelism of the track-oriented multiobject tracking algorithm at the procedure, or functional, level. The reasons for this choice of high-level parallelism over lower levels are threefold:

1. We wish to study the parallelism of the track-oriented approach as a single entity, not merely decompose it into a collection of random operations that may be implemented in a parallel fashion.

2. Once the high level parallelism has been identified, it will be possible to exploit the lower level parallel structure through methods available in the literature (e.g., matrix algebra).

3. The track-oriented approach is known to possess an extremely high degree of inherent procedure level parallelism, and it is this type of parallelism that distinguishes the track-oriented approach from other tracking methods.

The methodology employed in examining the structure of the track-oriented approach is to first decompose the algorithm into its functional units, with an explanation of the nature and the size (operation counts and memory

requirements) of these units. In addition to the tasks themselves, the dependency relations between individual tasks must be formulated. Dependency relations determine the ordering of the computational tasks, and so define their dependence or independence. The best expository method found for displaying these characteristics is that of a computation graph. A computation graph is a directed graph where the nodes represent some task in the algorithm, and the arcs represent dependency relations between source and sink pairs. By employing computation graphs it is possible to determine the intrinsic parallelism in the algorithm, independent of any specific computer architecture. This parallelism is described by measures such as the achievable speed-up and number of parallel tasks for each of the tracking functions.

Once the structure and extent of the track-oriented approach is well understood, a reasonable decision on possible computer architectures may be made. In order to assure generality in our results we assume generic computer configurations and do not investigate specific computer systems. We have chosen two general classes of computer structures to investigate: associative processors and multiprocessors. Associative processors are single instruction stream computers that employ content-addressable memories. This class of computers has been successfully applied to simpler single hypothesis tracking approaches in the past, but to our knowledge has never been applied to a tracking algorithm as complex as the track-oriented approach.

The second class of computer architectures investigated is that of multiprocessors, which are defined to be asynchronous processing units that can perform different computational tasks concurrently. This is more flexible classification than associative processors as far as the required algorithm

format is concerned, as the individual processors are not constrained to lock-step operation. It is this flexibility that motivates the inclusion of multiprocessors in this study, for they afford the greatest potential for exploiting heterogeneous functional parallelism. We will investigate possible implementation methods for the tracking algorithm on a generic multiprocessor configuration. These methods result in algorithm structures that take advantage of the more powerful asynchronous processing capabilities of multiprocessors.

For both of the computer architectures mentioned possible restructurings of the tracking algorithm that increase the amount of available parallelism are investigated. This step is extremely important for the tracking functions that possess non-obvious parallelism.

## 1.4 RESULTS

It has been found that the track-oriented algorithm displays an extremely high degree of functional parallelism for most tracking procedures, and that those tasks that appear at first to be inherently sequential may be restructured so as to increase their available parallelism. The functions of predicting target tracks forward in time, gating sensor returns against predicted values, and updating the tracks with the selected returns are all inherently parallel by target track. These tasks correspond to operations on individual branches of the hypothesis trees. Once the individual branches are complete, it is necessary to select the likeliest hypothesis to employ in track pruning and system analysis. In the standard algorithm (i.e., implemented on sequential computers), both clustering and global hypothesis selection is inherently sequential in nature as each possible hypothesis must be evaluated and compared in turn.

Various measures of parallelism were obtained from the analysis of the sequential form of the track-oriented multiobject tracking algorithm. A theoretical speed-up of 144:1 was determined for an "optimal" parallel implementation of the algorithm over the same algorithm on a sequential computer. Note that this does not assume any restructuring of the standard algorithm in order to increase the amount of exploitable parallelism. Without restructuring of the algorithm substantial processor inefficiences will result due to the large amount of time required to perform the sequential tasks. For example, it was found that the clustering function has approximately one one-thousandth the number of parallel tasks that the update function has. Obviously, if the number of available processors was matched to the exploitable parallelism in the update task, most processors would sit idle during the clustering step.

An overall measure of the concurrency available within the different steps of the algorithm is the parallelism ratio. For an algorithm with the same degree of parallelism at each step the parallelism ratio is 1, but for the unaltered track-oriented algorithm the ratio is .16. The low parallelism of the clustering and global hypothesis steps, along with their substantial processing requirements, "unbalance" the algorithm. It is for this reason that a considerable amount of effort is spent attempting to increase the parallel nature of these two tasks by restructuring their computations. How this restructuring is accomplished is highly dependent on the specific computer architecture under consideration, but it will be shown that the resulting procedure parallelism is generally quite high.

The track-oriented approach proves to be well matched to associative processors, as it is a collection of fully parallel functions (e.g., track prediction) followed by search oriented functions (e.g., global hypothesis

generation). It will be shown that the various tracking functions can all be implemented on associative processors, each with a substantial degree of parallelism. The functions that display track-level parallelism are the simplest to implement on associative processors. The two tracking functions that pertain to global hypothesis management, those of clustering and global hypothesis generation, are both studied in detail as they are complicated tasks that display little obvious parallelism, especially the synchronous parallelism required in associative processors. Several possible implementations of the clustering function are presented, with varying requirements on memory size and varying degrees of achievable speed-up.

Multiprocessors, due to their flexibility, are even more applicable to the tracking algorithm. The multiprocessor model that was employed was a fairly relaxed model without major restrictions on memory, processing, or interconnection networks. The functions of clustering and global hypothesis formation are again substantially restructured to exploit the capabilities of multiprocessors.

In summary, the track-oriented multiobject tracking algorithm, as it is currently implemented on sequential computers, possesses varying degrees of inherent functional parallelism. By restructuring the tracking procedures that display the lowest parallelism, it is possible to enhance the available parallelism to the level where concurrent processing is quite attractive and efficient. This is accomplished for both associative processors and multiprocessors. However, as should be the case in an initial study, the computer models employed herein were quite relaxed. A major restriction that was not considered in this research is the communication required between concurrent tasks, including access to common memory. More specific computer architectures

should be investigated to determine the performance achievable on actual hardware.

## 1.5 OVERVIEW OF REPORT

The remainder of this report is arranged as follows:

In Section 2 we provide a brief overview of parallel computer architectures. A description of the relevant parallel computers is necessary prior to any analysis and design of application algorithms. This discussion is not meant to be a complete survey, but should provide enough background for the remaining sections.

Section 3 includes descriptions of existing parallel tracking methods and computers that have been located in the open literature. As will be seen, these methods all employ fairly simple tracking algorithms, and are implemented on associative processors.

We next introduce and describe the track-oriented multiobject tracking algorithm in Section 4. We will not concern ourselves with the derivation and computational minutiae of the algorithm, as we need only its structure. Far greater detail is available in the references cited.

In Section 5 we analyze the track-oriented tracking algorithm, independent of any specific computer architecture. This is accomplished by functionally decomposing the algorithm into its component tasks and describing the actions and processing requirements of each. Determining the inherent algorithm parallelism then allows us to estimate its theoretical performance on abstract computer architectures. This performance is restricted by the form of the standard algorithm alone, and so provides a useful baseline measure.

Once the algorithmic structure of the track-oriented approach is analyzed, it may be applied to chosen computer architectures. In Section 6 we adapt the track-oriented multiobject tracking algorithm to associative processors. Due to the single instruction stream nature of associative processors this restructuring is quite involved, especially for the more sequential functions.

Finally, in Section 7, the track-oriented approach is adapted to multiprocessors. This is a simpler task than that of Section 6 due to the greater flexibility afforded by the multiple instruction streams.

# SECTION 2

# PARALLEL COMPUTER ARCHITECTURES

## 2.1 INTRODUCTION

The classification and description of parallel computers is a necessary foundation for the analysis and design of application algorithms. The reason for this is the specific computer architecture defines the type of computation that can best be exploited. As will be obvious in the sequel, no two parallel computers of separate design will support the exact same algorithm structure to the same extent. Therefore, when considering the creation of parallel forms of algorithms the computer architecture must first be defined to the level that sufficiently represents its strengths and weaknesses. This is in contrast to standard sequential computers, where a given program may normally be transported from one machine to another with little or no change to the algorithm itself.

In the research described in this report specific computer architectures have been avoided in favor of more general classes of computers. As the purpose of this project was to study the adaptation of a known algorithm (track-oriented multiobject tracking) to various parallel computer structures it was not considered useful to concentrate on the details of previously produced machines. Also, many parallel computers have been built to solve a particular problem, and so should not be arbitrarily assumed to be applicable to any given situation. Indeed, a more realistic approach for this research was

found to be to define the computer attributes that best support the algorithm, instead of fitting the algorithm to a given computer. These attributes must be chosen from those found to be available to the given computer classification, and to this end specific examples within these classifications will be presented.

The computing system classification scheme most commonly encountered is Flynn's classification based on the multiplicity of instruction streams and data streams [6]. Stream here denotes a sequence of items (instructions or data) as executed or operated on by a single processor. The magnitude of interactions between the instruction and data streams gives rise to four general machine organizations:

Single instruction stream-single data stream (SISD)
Single instruction stream-multiple data stream (SIMD)
Multiple instruction stream-single data stream (MISD)
Multiple instruction stream-multiple data stream (MIMD)

These four classifications are illustrated in Fig. 2-1. The single instruction stream-single data stream organization, also known as sequential computers, includes most conventional computers. Single instruction stream-multiple data stream computers include both array processors and the more specialized associative processors. The multiple instruction stream-single data stream class has no members, and appears to be of little real use. The fourth category, multiple instruction stream-multiple data stream computers, includes multiprocessors and multiple computer systems. The two classifications of interest in this study are SIMD and MIMD computers, and will be described more fully in the subsections that follow.

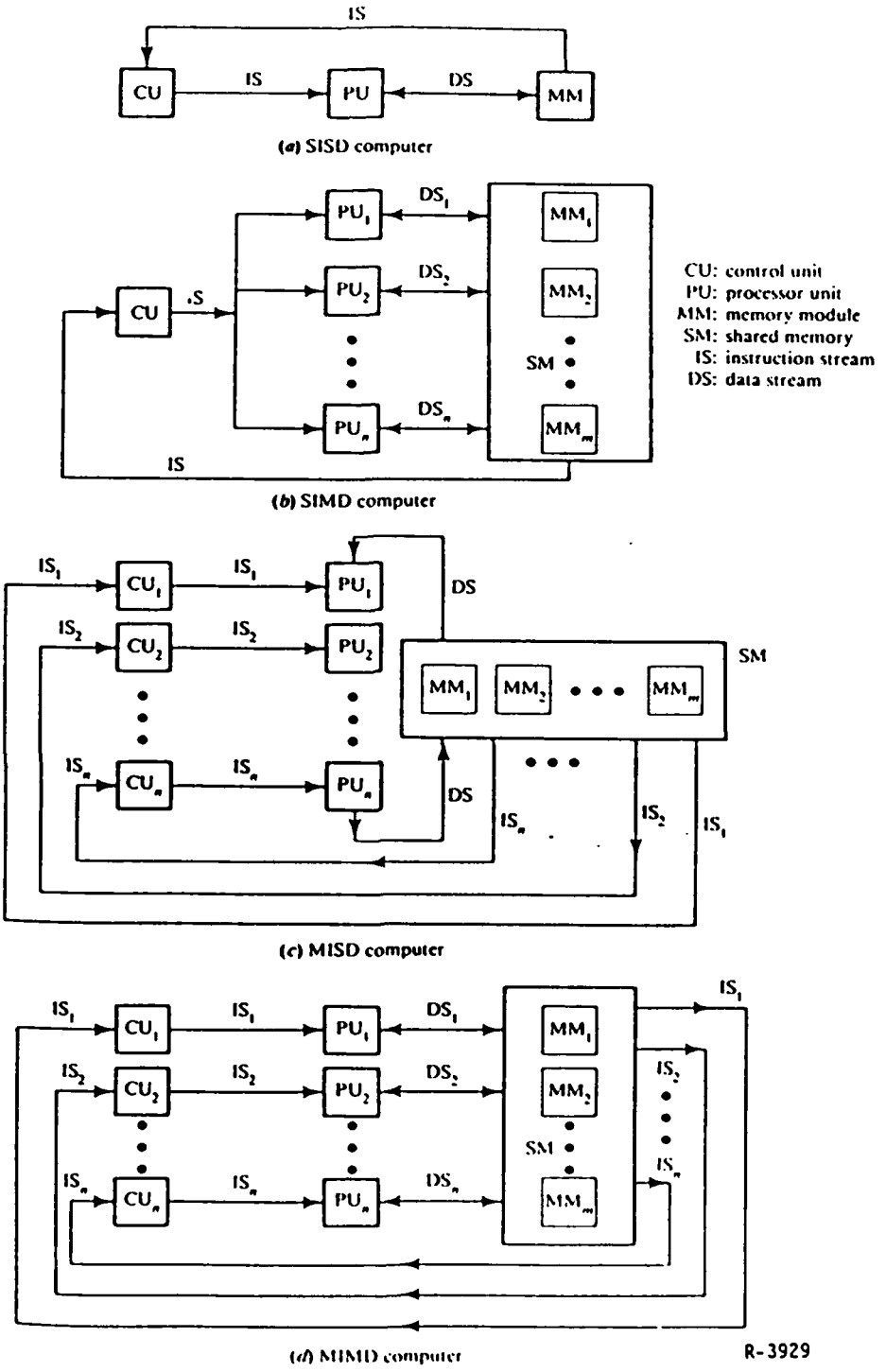



Figure 2-1. Classification of Computer Architectures [5]

## 2.2 SIMD COMPUTER ARCHITECTURES

### 2.2.1 Introduction

In single instruction stream-multiple data stream computers the parallel computations are performed in lock-step with the same master instruction being applied synchronously to separate data sets. This master instruction stream is broadcast by the control unit (CU) to all of the individual processing elements (PEs). Whether or not a particular PE executes the instruction is determined by a control mask, which either activates or disables the PE. Therefore, only a subset of the PEs performs the given computation or communication, the rest remain idle. To enable synchronous manipulation in the PEs, the data must be permuted and stored in the memory modules (MMs) in vector form.

SIMD computers may be subdivided into three basic categories: array processors, associative processors, and pipelined processors [6]. An array processor is an SIMD computer built around a conventional random-access memory (RAM), wherein memory words are accessed by their addresses. In contrast, associative processors contain an associative memory (AM) which is content addressable, allowing parallel access to multiple memory words based upon their values. Pipeline machines may be considered as time-multiplexed versions of array processors, assuming the individual elements of a vector are viewed as multiple data streams.

The inclusion of pipelined processors in the SIMD group is debatable. Some consider pipelining (most often likened to assembly lines in manufacturing) to be in the SISD category as there is actually a separate instruction for each operand pair, even though the same instruction is repeated over many operands [7]. Pipelining is a common method of exploiting the parallelism

available at the intra-instruction level. This involves decomposing the basic instruction into several subfunctions (instruction fetch, instruction decode, etc.) which can then be processed repeatedly in an overlapped fashion. The CDC 7600 and the IBM 360/91 were two of the earliest computers to employ pipelining of their arithmetic units. More modern machines include the Cyber 200 series and the well known Crays. As discussed in the opening section of this report, the low level parallelism available from pipelining, while of obvious significance in achieving system speed-up, is of less interest than the higher level parallelism imbedded in the algorithm structure itself. For this reason pipelining will not be considered further so that attention may be focused on the SIMD computers capable of higher level parallelism - array and associative processors.

The general configuration of an SIMD processor is shown in Fig. 2-2. One control unit provides the instructions to $N_p$ synchronized processing elements. Programs are stored in the CU memory, usually loaded from an external host computer. The CU interprets the program instructions, performing control and scaler instructions which cannot be converted into parallel form within the CU and broadcasting to the PEs the instructions to be executed in parallel. Participation of the PEs in the execution of the parallel instruction is determined by a control mask that either activates or disables the individual PEs. This selective activation of the PEs enables such features as data dependent conditional branching, where different sets of PEs follow different program paths. Unfortunately, the SIMD constraint that all units process the same instruction at a particular time precludes the concurrent execution of asynchronous tasks. In other words, the different tasks must be processed sequentially.

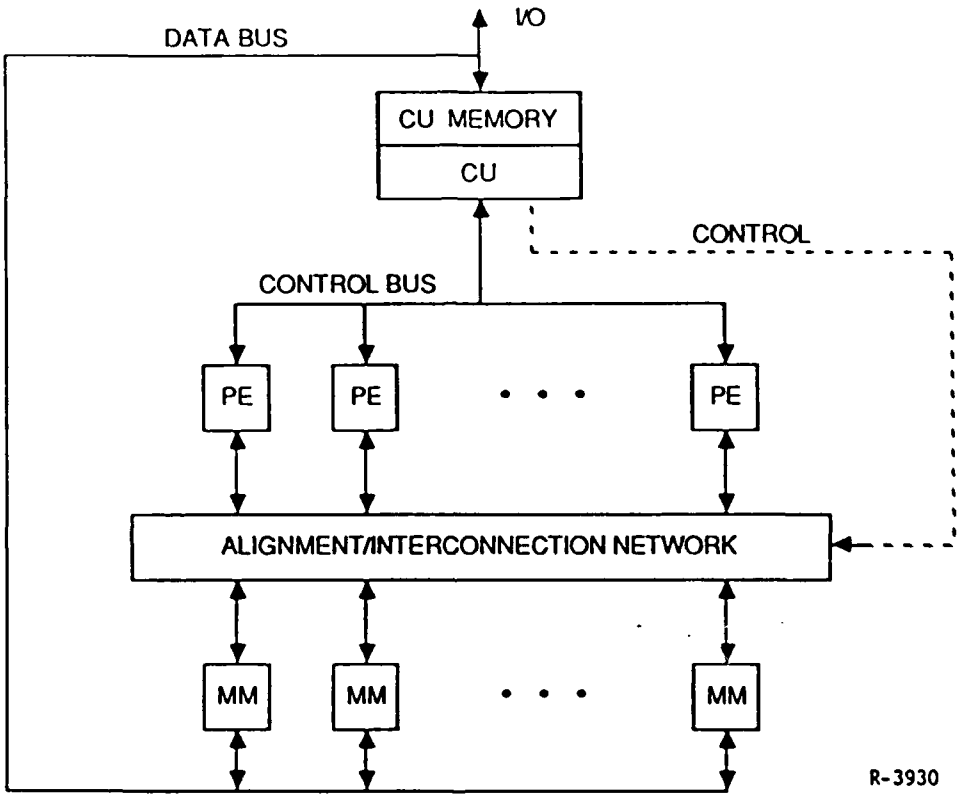



Figure 2-2. General Configuration of SIMD Processors

Each PE consists of an arithmetic and logic unit (ALU) with (typically) its own registers and local memory. Instructions broadcast by the CU are executed in the PEs on different component operands fetched from the memory modules. The PEs do not decode instructions themselves, but merely accept instructions transmitted by the CU. The degree of complexity of the individual PEs is a crucial determinant of processing capability. In a SIMD computer the PEs may be able to process either bit operands or word operands, depending on the design. A string of bits, one from each word, is termed a bit-slice processor, since each PE is operating on bits taken from the same location of the operands. Word-slice processing occurs in IMD machines that can process a word at a time. The Goodyear Aerospace Massively Parallel Processor (MPP) [8] is an example of a bit-slice arracy processor containing 16,384 bit-serials PEs. An example of a word-slice array processor is the ILLIAC IV, capable of handling 64-bit words in each of its 64 processors.

The alignment/interconnection network (A/IN) shown in Fig. 2-2 defines both the communication between PEs and from the PEs to the $N_m$ memory modules. The complexity and flexibility of this network differs among architectures. Various interconnection networks have been proposed for and implemented in SIMD computers [10], [11] for communicating among the PEs and MMs, and also between the PEs and MMs. Such networks are necessary for the PEs to cooperate on a common problem by exchanging or sharing both data and results. Inter-PE communication is usually of greater concern than processor-memory communication in SIMD computers as most often each PE is hard-wired to its own local memory. An example of such a machine is the MPP [8] in which each of the PEs is associated with a 1024-bit RAM.

The inter-PE network topology can be depicted as a graph whose nodes represent switching points (not necessarily PEs) and whose edges represent communication links. Figure 2-3 shows examples of static network topologies where the nodes are equivalent to PEs. Static network topologies have predefined connections. If two PEs are not directly connected they can communicate only through intermediate PEs in a sequence of steps. Dynamic networks allow reconfiguration of the data links for greater flexibility of communication. Alignment networks get rapidly more complex as the number of PEs increase, therefore the PEs are usually limited to a small number. A more general A/IN that allows memory modules to be shared by the PEs through the alignment network is the Burroughs Scientific Processor (BSP) [9]. The BSP contains a path-switching alignment networks to connect its 16 arithmetic elements to each other and to its 17 common memory modules. The discussion of interconnection networks is beyond the scope of this report and is provided in more detail in the references sited.

### 2.2.2 Array Processors

An array processor is a synchronous parallel computer that operates in a lock-step fashion in which data items are stored in a random access memory (RAM). The previous discussion covers the major portions of an array processor function, with the exception of memory access. The CU broadcasts a global memory address to each PE. This memory address specifies the location of the data within the individual PE's memory module. Each PE may offset the broadcast address by its own index register so that different locations in different MMs can be accessed simultaneously with the same global address. Operand locations may also be specified by the CU by broadcasting the PE registers to be used.

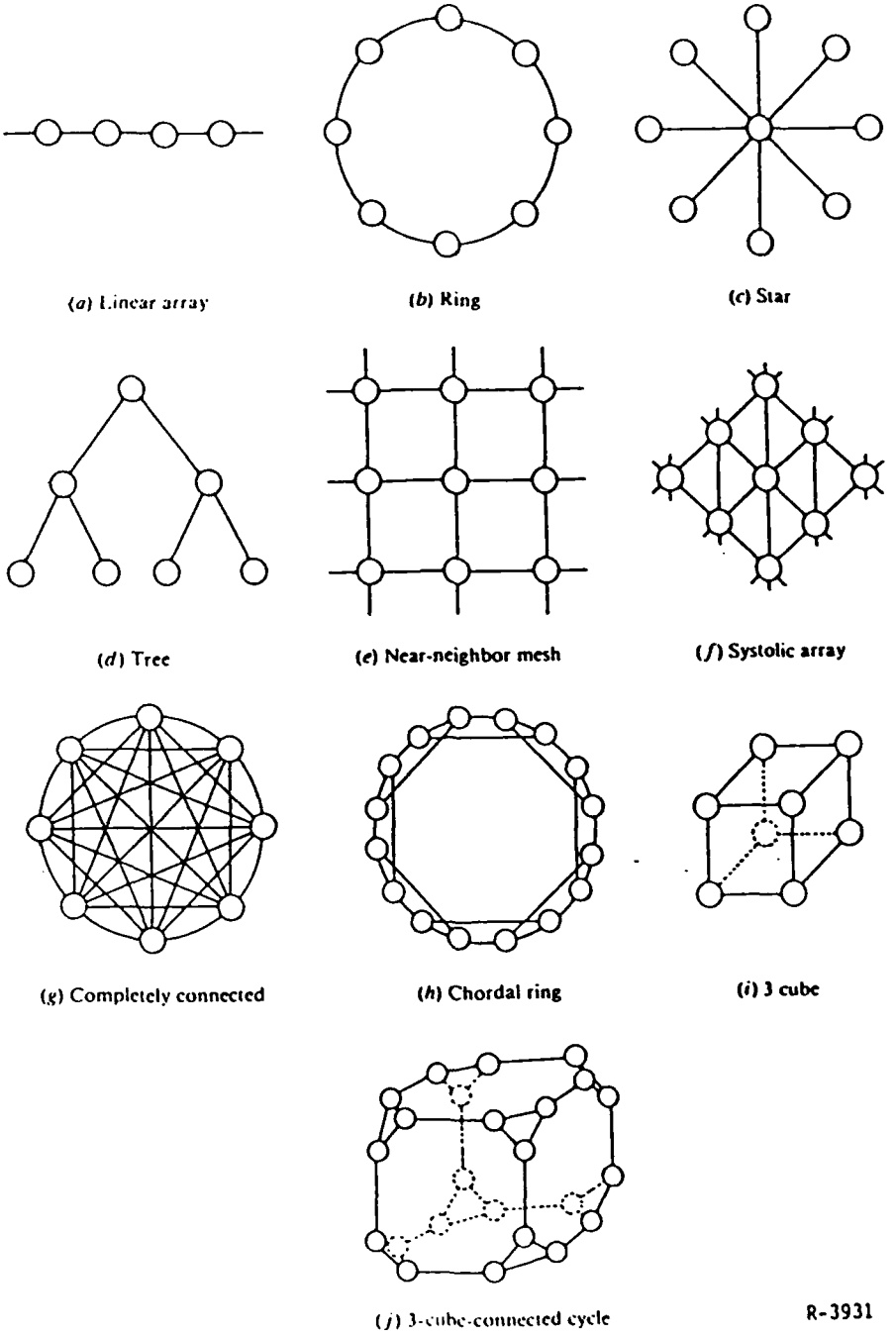



Figure 2-3. Static Interconnection Network Topologies [11]

An example of an array processor is the Goodyear Aerospace Massively Parallel Processer designed for NASA to solve two-dimensional data processing problems such as satellite imagery. The MMP is comprised of 16,384 microprocessors configured in a 128 by 128 square array. The PEs are bit-serial processors capable of handling variable length operands. Each PE has a private RAM of 1024 bits. Inter-PE communication is accomplished via a nearest-neighbor topology (c.f. Fig. 2-3e). More flexible routing topologies were considered but were decided against due to the complexity inherent in such a large number of PEs.

The MMP basic structure is shown in Fig. 2-4. The array unit (ARU) contains the 128 by 128 square of processing elements, along with an extra 128 by 4 rectangle of PEs used as spares to reconfigure the ARU in the event of a fault. The local memory of 1028-bits per PE can conceivably be expanded to 65,536-bits per PE (i.e., the control unit generates 16-bit memory addresses). The ARU is organized with a 1028 planes of 16,384 elements each, and can handle data of arbitrary precision. Two-dimensional planes were chosen over other organizations (such as words) due to the MMP's design purpose of two-dimensional data processing. In general, an array of 16,384 words with n bits per word is stored on n planes of the ARU. This large number of PEs, along with the ARU cycle time of 100 ns, gives the MMP a very high processing rate as shown in Table 2-1.

Inter-PE communication occurs in a nearest neighbor (north, east, south, west) fashion. At the array edges links can either be left open or connected to the opposite edge. These routing choices may be used to change the square array into such topologies as a one-dimensional line, a circle, or a torus. Shifts occur in one of the four nearest neighbor directions, and are either

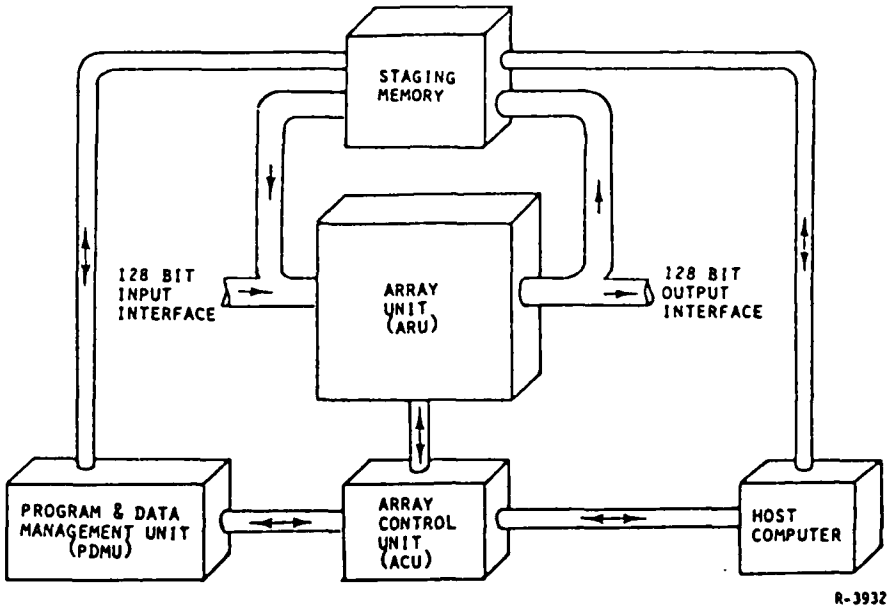



Figure 2-4. Block Diagram of Massively Parallel Processor [8]

TABLE 2-1. MMP PROCESSING SPEED [8]

| OPERATION | PROCESSING SPEED |
|---|---|
| Addition or subtraction: | |
| 8-bit fixed-point (9-bit result) | 6553 |
| 12-bit fixed-point (13-bit result) | 4428 |
| 16-bit fixed-point (17-bit result) | 3343 |
| 32-bit floating-point | 470 |
| Multiplication: | |
| 8-bit fixed-point (16-bit result) | 1861 |
| 12-bit fixed-point (24-bit result) | 910 |
| 16-bit fixed-point (32-bit result) | 538 |
| 32-bit floating-point | 291 |

masked (only those PEs with a 1 in their mask register participate) or unmasked (all PEs participate) operations.

The array control unit (ACU) performs scaler (single data items) arithmetic and controls the PEs. The ACU is made of three sections that can operate in parallel: PE control, I/O control, and main control. The PE control unit controls operations in the processing planes of the ARU; I/O control manages the flow of data in and out of the ARU; and the main control runs the main MMP application program and performs all scaler operations.

The program and data management unit (PDMU) is a back-end minicomputer that controls the overall flow of programs and data in the system. It manages data flow in the array, loads programs into the controller, executes system test and diagnostic routines, and provides development facilities.

The staging memories reside between the I/O ports of the ARU and the front-end computer (PDMU or host), and can buffer and re-format arrays of data. Reformatting is usually required because the PDMU or host typically operates on an array of data in a word serial mode while the ARU receives and transmits the array in a bit serial mode. The staging memory has a capacity of 2 megabytes and a transfer rate of 20 megabytes per second.

The MMP was built specifically for image processing tasks, but is sufficiently flexible to be applied to many other large-scale problems. Image processing applications include feature extraction, pattern classification, and scene analysis. Other suggested applications encountered are fluid dynamics and database management.

### 2.2.3 Associative Processors

An associative processor consists of a storage device wherein stored data items can be accessed or retrieved on the basis of their content (associative memory), along with the capability to perform data transformations upon more than one set of arguments with a single control instruction (parallel processor) [13], [14]. As with all SIMD computers, the parallel computations in an associative processor are performed in lock-step with the same instruction being applied synchronously to separate operands.

The primary difference between associative memory (AM) and the more common random-access memory used in other processors is that RAM is accessed via the word address, while AM allows for the parallel access of multiple memory locations by specifying the word content, not its location. The main advantage to associative processors over array processors is the ability to perform search, comparison, and logic operations over all words in the memory array in parallel. Note that it is also possible for AMs to be accessed by address as well as content in some systems, albeit in a slightly indirect manner.

In order for the AM to accomplish its task it is necessary for sufficient logic to be available at each word to determine whether that location contains data that match some globally defined criterion. The complexity of this logic organizes APs into two categories: bit serial and bit parallel. In bit serial APs one bit of each memory word is acted upon in parallel (word parallel), with subsequent bits of each word handled sequentially. In bit parallel APs the processing is performed both parallel by word and parallel by bit. Bit parallel APs will exhibit higher processing speed, but they suffer from more complex detection logic than bit serial architectures and are therefore more expensive to build.

A generic associative processor architecture is shown in Fig. 2-5. In itself this schematic differs little from conventional array processors. We can, for the purposes of this study, ignore specific global architecture characteristics and focus on the general associative memory and its operation. Specific input/output considerations will be presented as they impact the performance of high I/O algorithms such as multiobject tracking.

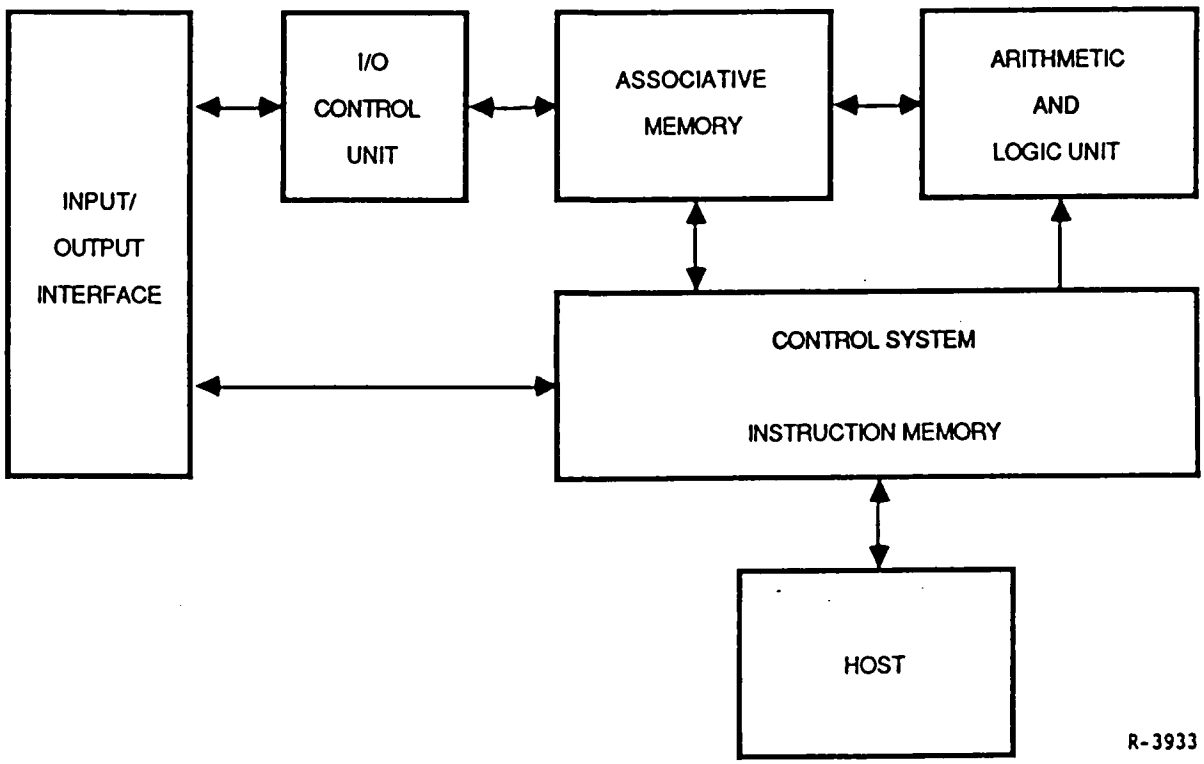



Figure 2-5. Generic Associative Processor Architecture

Figure 2-6 shows the structure of a basic associative memory. The AM consists of W array rows (also called words or cells) with B bits per row. A contiguous subset of the array beginning at a given bit and having a given length, in each word, is termed a field. A field with length 1 is called a bit-slice.

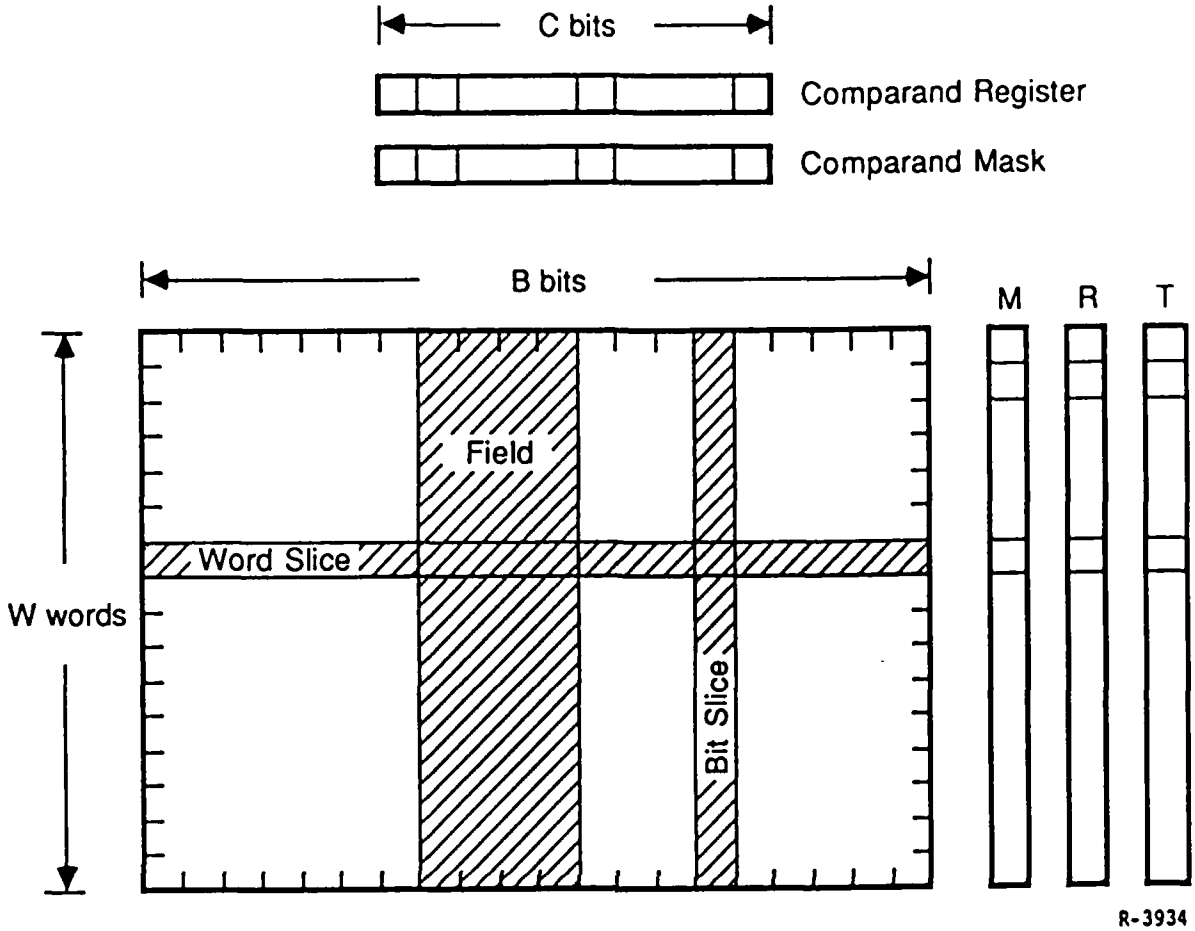



Figure 2-6. Associative Array Memory

For bit serial APs there will be one processing element for each word of memory, and these PEs will operate on a bit-slice of data at a time. Bit parallel APs will have a more complex PE for each word and are able to process the whole array in parallel. The common datum value for use in comparisons is stored in the comparand register. In some systems the comparand contains as many bits as an array word, in others it contains much less. For large comparands it is usually necessary to include a comparand mask register to enable or disable bit slices of the array that are to be involved in the comparand operation. This allows for parallel operations in more than one field of a

word. Smaller comparands may not employ a mask, but instead specify the field to be compared. We will assume the latter case in the sequel.

The three registers beside the array are the Mask (M), Response (R), and Temporary (T) registers (also called M, X, Y). These are all dimensioned by the number of words in the array. The mask register is used to enable or disable words to participate in operations (searches, logic, I/O, arithmetic, etc.). The response register holds the result of the most current associative search. The temporary register is a storage device. Any of the registers can be used in logical operations with each other to alter their values. This allows, for instance, the results of several searches to be combined in one register.

The contents of the comparand may be compared with those of the array words via the following operations (searches):

| | |
|---|---|
| EQUAL | NOT EQUAL |
| GREATER THAN | GREATER THAN OR EQUAL |
| LESS THAN | LESS THAN OR EQUAL |
| INSIDE OR EQUAL LIMITS | OUTSIDE LIMITS |

If the result of the search in a particular word is true, the corresponding response bit is set. In addition to the above, some systems support the capability of using an array field instead of the comparand (i.e., each word uses the field in that word) as well as the ability to find the minimum or maximum of a field. As the complexity of these operations increase so will the computation time.

Logic operations are bit-wise logical combinations of an array field with other array fields or common register, with the result entered into an array field. AND, OR, and XOR (exclusive OR) are assumed to be available.

Associative arithmetic functions allow two array fields, or an array field and the common register, to be arithmetically combined. Most lower level functions are found on every AP architecture.

Data movement, either within the array or between the array and the external system (I/O), can be managed in several ways. Data can be input or output to the array through the comparand register. On input to the AM the contents of the comparand can be written to some or all of the words in parallel depending on the setting of the mask bits. Output to the comparand may be handled by bit-wise logical OR-ing the fields of all the active (unmasked) words on the data channel, so that the value reaching the comparand is a combination of the output fields. Other systems may only allow one word to be active at a time on output, so the logical OR-ing does not occur. We will employ logical OR-ing of the output data later on and so assume its implementation.

Another method of data I/O is to include, as shown in Fig. 2-5, a data path directly to the AM. Depending on the design the I/O control unit may be capable of content addressable input (i.e., input only to words that satisfy some condition). Also, the data may be manipulated, or "flipped", so that either one word or one bit per word may be input. This is especially useful in bit serial APs since they normally operate on a bit-slice at a time, whereas data usually arrives in word format. Another useful feature is that, since the control and I/O are separate units, it is possible to have I/O and computation going on concurrently.

Data transfers within the array can be thought of as either "horizontal" shifts (field to field) or "vertical" shifts (word to word). Horizontal shifts are typically handled in one instruction using the field locations.

Vertical shifts in parallel require specialized hardware channels to get the high bandwidth transfers needed.

An example of an associative processor is the Goodyear Aerospace STARAN [15]. STARAN is a bit-serial AP consisting of up to 32 associative array modules. Each array module contains 256 one-bit processing elements along with a 256-word 256-bit multidimensional access (MDA) memory. A typical STARAN block diagram with four array modules is shown in Fig. 2-7. As the array modules define the processing capability of STARAN they will be described prior to the system architecture.

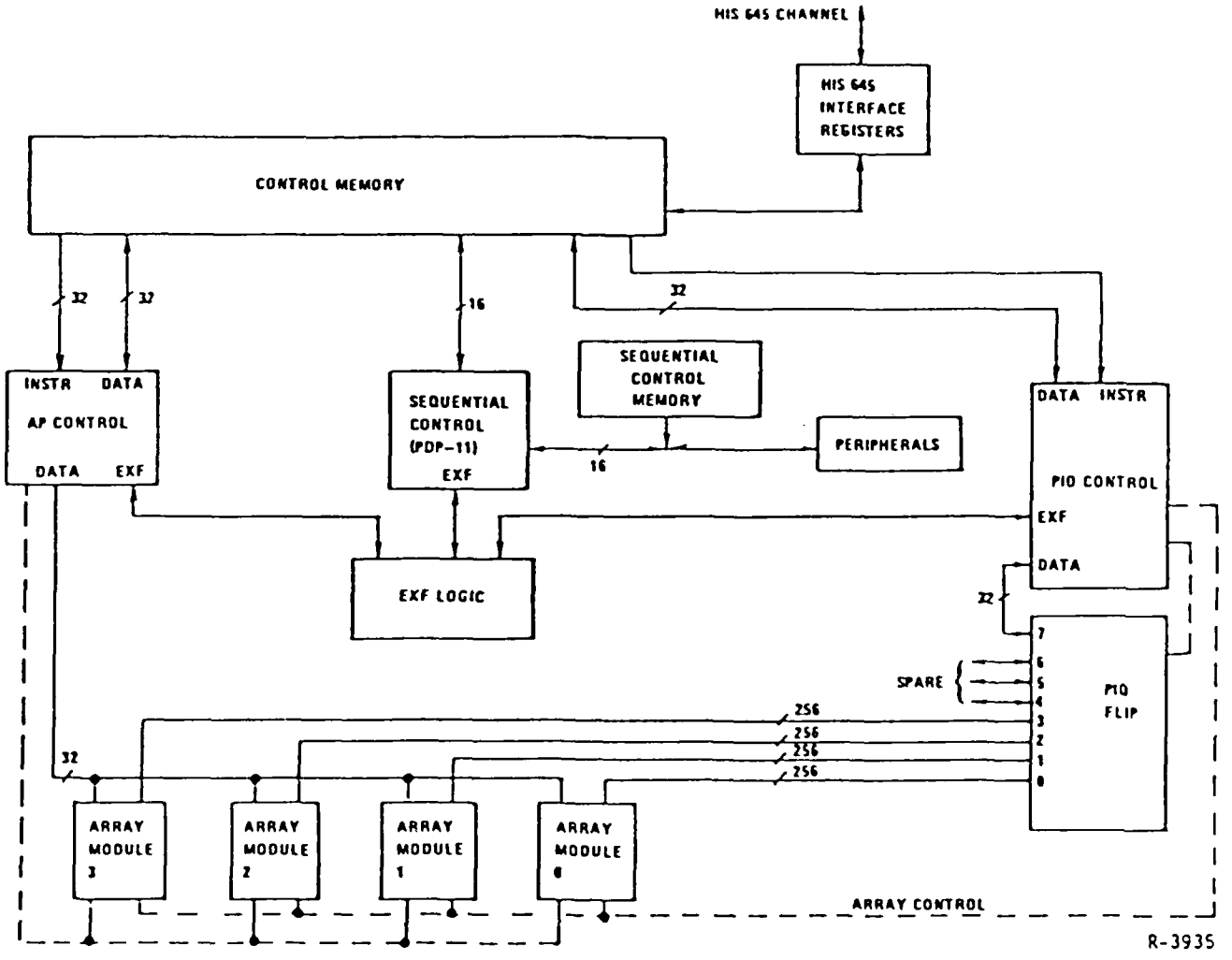



Figure 2-7. Typical STARAN Block Diagram [15]

Figure 2-8 shows the STARAN array module functionality. Each array module contains an MDA memory communicating with three 256-bit registers (M, X, and Y) through a flip (also called scramble/unscramble) interconnection network. In effect, each PE in the module contains one bit of each of the registers. The flip network [16], [17] may shift or permute bits in the source item such that words and bit-slices, as well as other templates, can be accessed easily. Permutations of the flip network also allow inter-PE communication, either by one PE reading directly from another PE's register, or indirectly from the MDA memory.

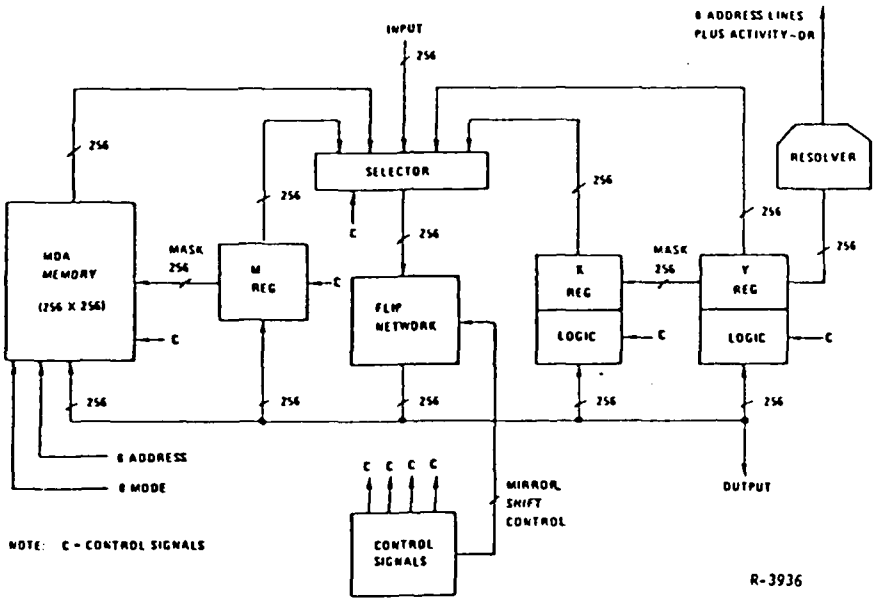



Figure 2-8. STARAN Array Module [15]

The multidimensional access memory is so named for its ability to accommodate both bit-slice access (for associative processing) and word-slice access (for I/O). The 256-bit wide read and write buses allow simultaneous access to one bit from each word or all the bits in one word, or a combination

of the two. It is also possible to format the data within the MDA in structures other than 256 words with 256 bits in each word. For example, the data may be arranged in 32 records, each of which contains 256 8-bit fields.

As shown in Fig. 2-7, each array module may be controlled by the AP control or the parallel input/output (PIO) control. Each array module contains an assignment switch that connects its control inputs and data buses to one of the two controls.

The AP control receives instructions from the control memory and can transfer data to and from the control memory, all over 32-bit channels. Data may also be exchanged with the array modules, but only over 32-bits of the 256-bit wide array I/O ports. The flip network facilitates communication to any part of the array module. The AP control contains the 32-bit comparand register, along with other instruction registers, pointers, and counters.

The parallel input/output unit is designed for high bandwidth I/O and inter-array data transfers, as well as performing processing and I/O in parallel. I/O ports to the PIO flip network allow any device (parallel-head disk stores, radar video channels, etc.) to communicate directly to with the array modules. Inter-array communication is handled by permuting data between array module ports. As each array module may be controlled either by AP control or by PIO control, it is possible for some arrays to be processing data while others are performing I/O.

The control memory holds both AP and PIO control programs. The main program usually resides in core memory (up to 32k words) with microprogram subroutines stored in bipolar memory.

The sequential control handles peripherals, controls the system from console commands, and performs diagnostic tests.

External function (EXF) logic serves to synchronize the AP control, PIO control, and sequential control.

Applications of STARAN include radar tracking, (discussed further in subsection 3.3), fast Fourier transforms, and sonar post-processing.

## 2.3 MIMD COMPUTER ARCHITECTURES

### 2.3.1 Introduction

In multiple instruction stream-multiple data stream computers several processing elements operate concurrently in an asynchronous manner under integrated control to accomplish some desired task. This definition serves to distinguish MIMD multiprocessors from single instruction stream machines (SISD and SIMD) which obviously cannot operate asynchronously, and from multiple-computer systems or networks which consist of several separate and discrete computers that are not controlled by a central unit. The class of MIMD computers may be further defined by the following characteristics [18]:

- A multiprocessor contains two or more processors of approximately comparable capabilities.
- All processors share access to common memory.
- All processors share access to input/output channels, control units, and devices.
- The entire system is controlled by one operating system providing interaction between processors and their programs at the job, task, step, data set, and data element levels.

One of the most standard classifications of MIMD systems is based upon how the individual processing elements communicate with each other. The two architectural categories most often encountered are those of tightly coupled and loosely coupled systems [5]. In tightly coupled multiprocessors the PEs communicate indirectly through a shared main memory. In loosely coupled

multiprocessors the PEs exchange data directly through a message passing system. Both of these approaches are discussed further below.

### 2.3.2 Tightly Coupled Systems

The tightly coupled, or shared memory, category may be the most common multiprocessor architecture. Figure 2-9 is a typical model of a tightly coupled system (TCS). It consists of p processors, 1 shared memory modules, and d I/O channels. The units are connected through a set of three interconnection networks. The processor-memory interconnection network (PMIN) is basically a switch that can connect every processor to every memory module. This switch may be either a p by 1 crossbar switch matrix, a multiported memory, or a multistage network. Shared memory is employed by the individual processors much like a blackboard: any processor can write information on it, which any other processor can subsequently read. If a processor wishes to communicate with another processor, it simply writes the message into the shared memory. Unfortunately, a memory module can only satisfy one processor's request in a given memory cycle. Memory contentions will therefore result whenever two or more PEs try to access the same memory unit concurrently. To avoid excessive conflicts the number of memory modules is usually as large as the number of PEs. Also, a small local memory or high speed cache may exist in each PE to help alleviate this problem. The unmapped local memory (ULM) in Fig. 2-9 provides this capability.

Models of memory access in tightly coupled multiprocessors used for algorithm development typically take one of three forms: exclusive read-exclusive write (EREW), concurrent read-exclusive write (CREW), and concurrent read-concurrent write (CRCW). In the EREW shared memory model all references to a module in the common memory must be made exclusively. The CREW model allows

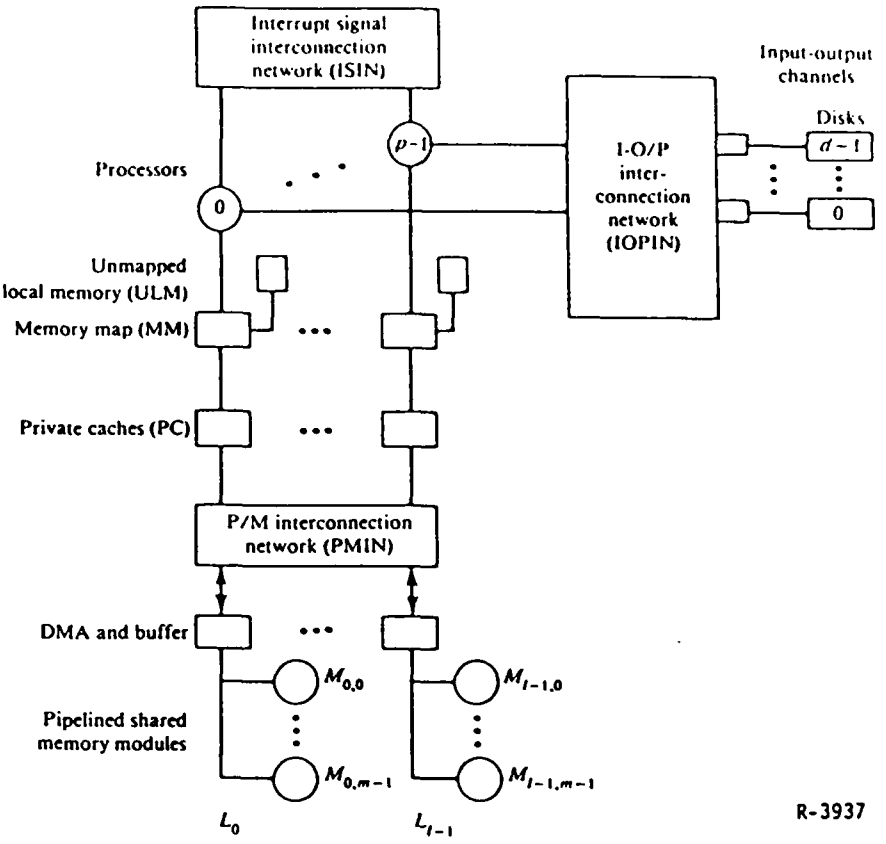



Figure 2-9. Tightly Coupled Multiprocessor System [5]

concurrent read operations but writes must be performed exclusively. Finally, the CRCW model permits both concurrent read and write operations. While the EREW model is the most realistic, the CRCW and CREW models are more prevalent in the algorithm literature.

As the individual processors must pass data through the main memory, the inter-PE data communication rate is on the order of the bandwidth of the concurrent read operations but writes must be performed exclusively. Finally, the CRCW model permits both concurrent read and write operations. While the memory. This provides a further inducement to include a high-speed buffer (cache) between the processor and main memory.

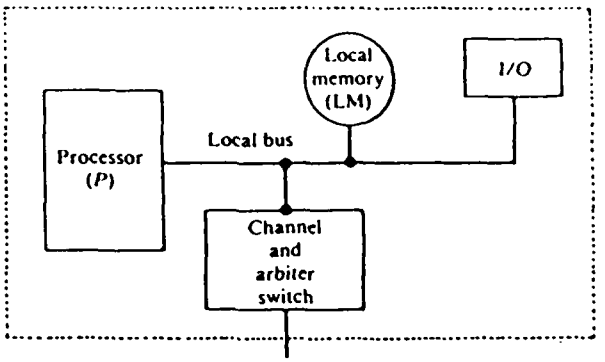
The second interconnection network in the TCS is the interrupt-signal interconnection network (ISIN). The ISIN allows each processor to interrupt the execution of any other processor, thereby facilitating synchronization between PEs. Note that messages are not sent via the ISIN, only interrupts. The final interconnection network is the I/O-processor interconnection network (IOPIN), which permits a PE to communicate with an I/O channel.

Examples of tightly coupled systems include the Carnegie Mellon University C.mmp [5] and the more recent Butterfly parallel processor by BBN Laboratories [19]. The C.mmp architecture consists of 16 processing elements and 16 common memory modules connected via a 16 x 16 crossbar switch. The Butterfly is a 128 processor shared memory system. Processors make memory references through a Butterfly switch. The switch is a Banyon network that uses 4 x 4 switch elements in its implementation.

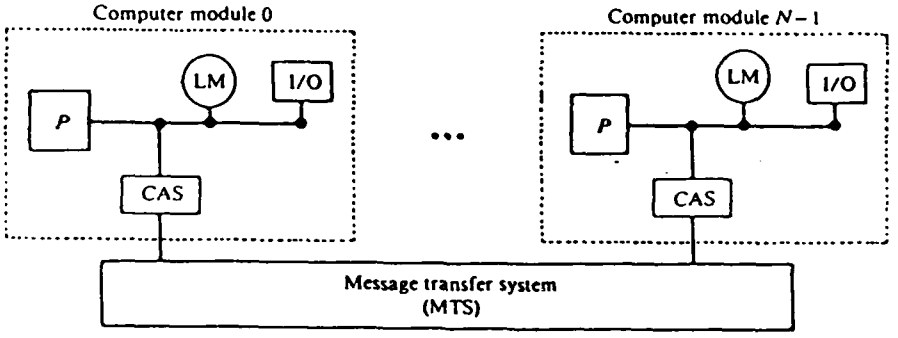
### 2.3.3 Loosely Coupled Systems

In loosely coupled systems each processor generally has a substantial amount of local memory where it accesses most of the instructions and data. The concurrent processes resident on the PEs communicate through a message transfer system (MTS). The topology of this interconnection network is the dominant characteristic in determining the degree of coupling in the system. Typically, loosely coupled systems provide good throughput at the expense of flexible resource sharing, while tightly coupled systems provide flexible resource sharing at the cost of degraded throughput. Therefore, loosely coupled systems are usually more efficient when the interactions between tasks are minimal.

Figure 2-10 is an example of a loosely coupled computer architecture. An individual computer module is shown in Fig. 2-10a. This module consists of a processor (P), local memory (LM), I/O, and a channel and arbiter switch (CAS). The CAS is responsible for choosing among modules whenever tow or more attempt to access the same physical segment of the MTS.




(*a*) A computer module




(*b*) Loose coupling of computer modules

Figure 2-10. Nonhierarchical Loosely Coupled Multiprocessor System [5]

The message transfer system may be a time-shared bus, a shared memory system, or any other number of connection networks (c.f. Fig. 2-3). We will not discuss interconnection networks here, and direct the reader to the references, particularly [20].

## 2.4 CONCLUDING REMARKS

As the research presented in this document is an initial attempt at applying the track-oriented multiobject tracking algorithm to multiprocessor architectures it behooves us to keep the discussions as general as possible. Also, as an initial effort, it is felt that the emphasis should lie on the algorithm far more heavily than any specific computer system. For these reasons we have restricted our discussions of parallel computer systems to be both brief and generic. More effort was spent in presenting SIMD architectures as they are more restrictive, and therefore require a more involved explanation. MIMD architectures are a far more general class. A multitude of information far more detailed than presented (or needed) here is available in the references supplied.

# SECTION 3

# EXISTING PARALLEL TRACKING METHODS AND COMPUTERS

## 3.1 INTRODUCTION

Parallel processors have been applied successfully to multiobject tracking problems in the past. The complexity of the tracking algorithms implemented on these computers has been limited, though, most having been developed in the early to mid-1970s. Tracking methodologies and computer architectures of two major parallel tracking approaches, PEPE and STARAN, will be discussed in detail in subsections 3.2 and 3.3, respectively. These two systems dominate the open literature, both in parallel tracking and in associative processors. Subsection 3.4 contains a discussion of the Airborne Associative Processor (ASPRO) which, like STARAN, was developed by Goodyear Aerospace for tracking applications. The Associative Linear Array Processor (ALAP), from Hughes Aircraft, is described in subsection 3.5. It is interesting to note that all of these parallel tracking computers are associative SIMD processors, and that all employ single hypothesis (or simple track splitting) tracking methods.

## 3.2 PEPE AND DERIVATIVES

The Parallel Element Processing Ensemble (PEPE) [21]-[28] is a highly parallel experimental associative processor designed to address the ballistic missile defense (BMD) problem through augmentation of a general purpose

sequential computer. PEPE is a fully parallel, or word-parallel and bit-parallel, AP. The PEPE architecture allows for overlapped input/output and arithmetic functions. In addition, data transfer with external devices may take place simultaneously with the transfer of data into or out of the PEs. Its design has evolved from work done at Bell Laboratories in the early 1960's on distributed logic configurations for APs. In distributed logic APs the comparison logic is associated with each character cell of a fixed number of bits or with a group of character cells. Such a configuration is more economical than word-organized APs, where the comparison logic is associated with each bit cell of every word [5].

A block diagram of the PEPE system architecture is given in Fig. 3-1. The host computer is a CDC 7600 and the test and maintenance (T&M) computer is a Burroughs B1700. The PEPE configuration consists of a control system and up to 288 processing elements arranged in bays of 32 PEs each. Each PE may simultaneously respond to commands from each of the three control units - the Correlation Control Unit (CCU), the Arithmetic Control Unit (ACU), and the Associative Output Control Unit (AOCU). PEs can be added or disconnected without effecting the operation of the system. This is due to both the content addressing characteristic of APs (since PEs are activated or deactivated based on data comparison, not address) and the lack of direct connections between PEs. If data has to be transferred between PEs it will have to be routed through the host [29].

The basic configuration of the PEPE processing elements is shown in Figs. 3-1 and 3-2. Each PE consists of 3 units: an Arithmetic Unit (AU), Associative Output Unit (AOU), and Correlation Unit (CU) sharing an Element Memory (EM). Each computational unit (AU, AOU, and CCU) contains a one bit activity

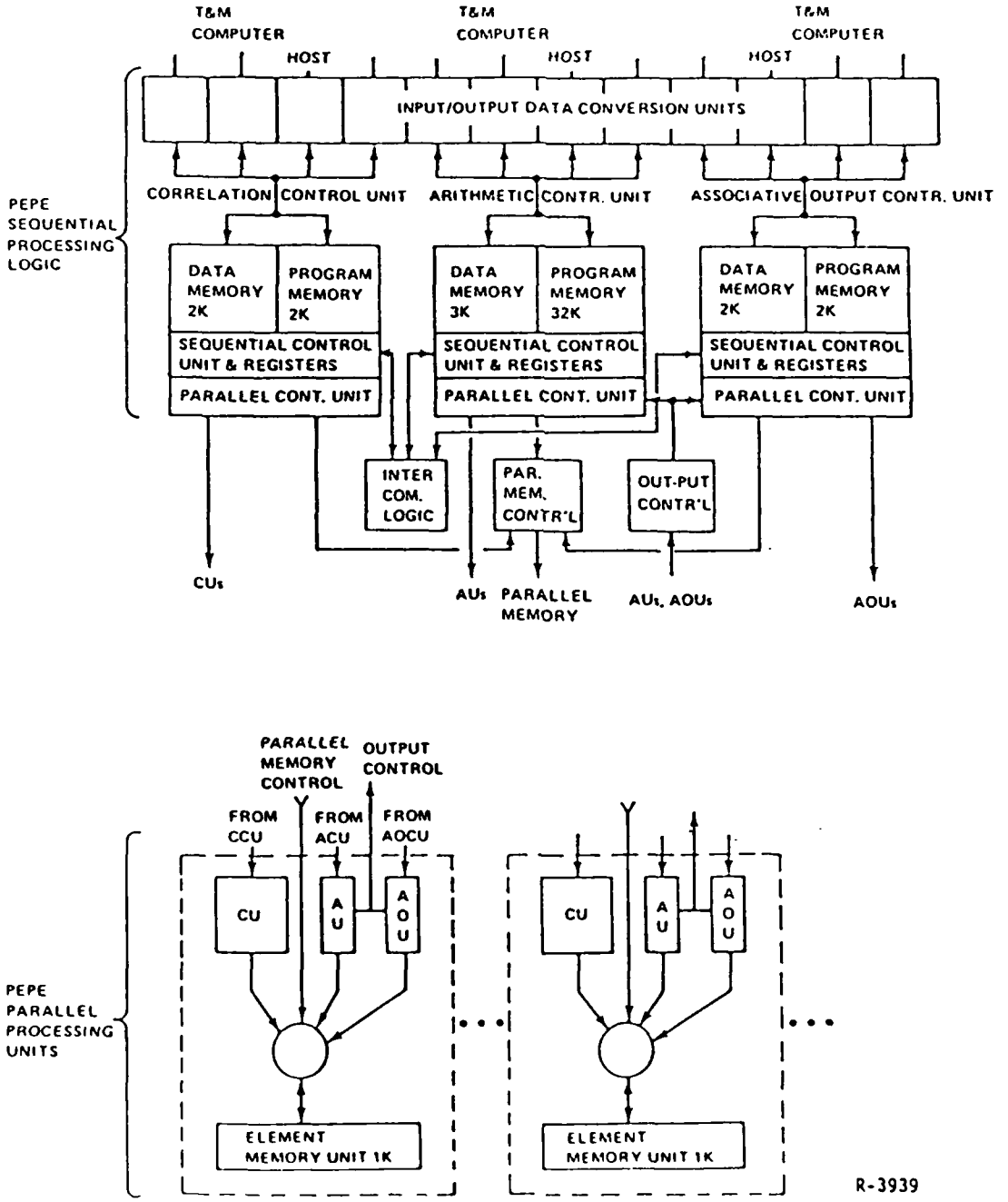



Figure 3-1. PEPE Architecture [21]

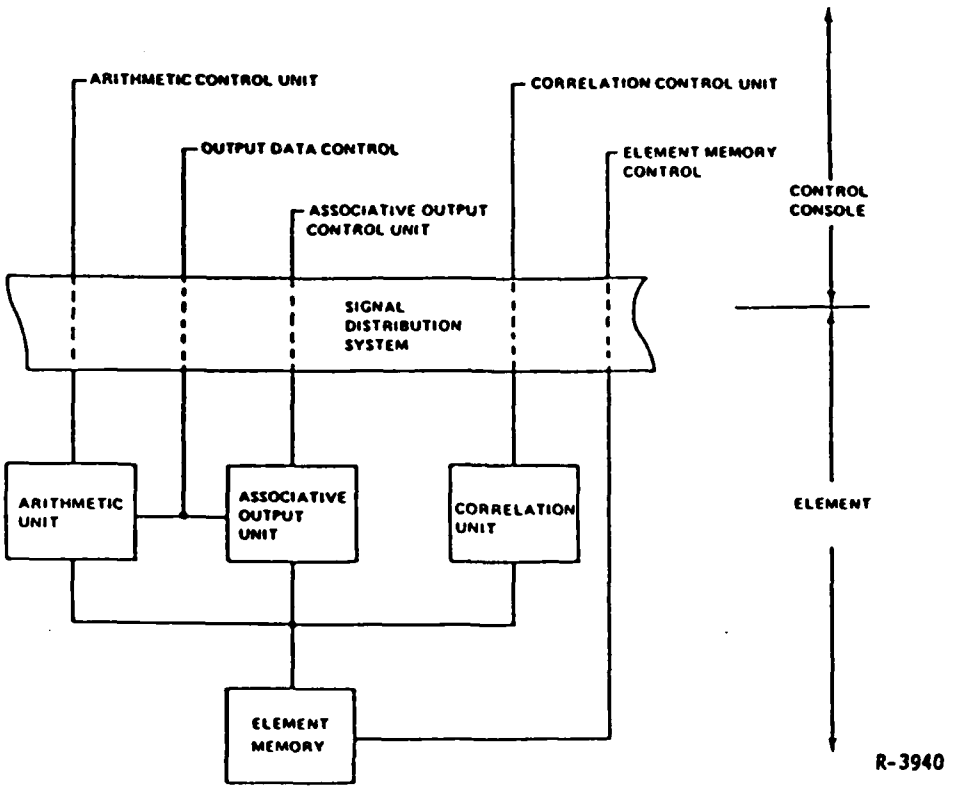



Figure 3-2. PEPE Processing Element [21]

register which is used to determine if the individual PE will respond to a parallel instruction. Since all three units may operate concurrently, a 288 element PEPE can effectively perform 864 simultaneous instructions. If there are PE memory conflicts this number will be lower.

Parallel integer, logical, and floating point operations are carried out in the AUs, under control of the ACU. Input to the PEs takes place through the CUs from the CCU. Each CU contains 16 registers which support parallel integer and logical instructions, and register-to-register correlation operations. The associative nature of PEPE is due to the ability of each CU to correlate data broadcast by the CCU with the data in its registers. The AOU

allows data to be output from the PEs to the output control in an associative manner. The PE memory consists of 1024x32 bit ECL storage. All EMs receive identical information from the Element Memory Control during execution of a particular function.

The PEPE control console (Fig. 3-3) contains the ACU, AOCU, and CCU which provide the instruction execution control for PEPE. The ACU manipulates the parallel database in the PEs; the AOCU outputs data from the parallel database; and the CCU inputs new data into the parallel database. The control console also contains functional units to support the following operations:

- Inter-control unit interrupts
- Error recovery
- Processing element output
- Element memory conflict resolution
- Maintenance and diagnostic tasks
- Input/Output data conversion.

Figure 3-4 shows the functional configuration of the three control units. Each unit has its own program and data memory, as also shown in Fig. 3-1. Programs may consist of a combination of sequential instructions (executed in the Sequential Control Logic) and parallel instructions (executed in the Parallel Instruction Control Unit). The ACU has a relatively slow cycle time of 200-300 ns due to its large (32k) program memory. Because of this the ACU parallel instructions are routed through a 16-step queue prior to execution.

The input/output units (IOUs) are included in each control unit to provide for data transfer to and from the host and T&M equipment. In addition to control unit start/stop and interrupts, the IOUs are capable of block data transfer to and from the control units initiated by either the host or the sequential control. The IOUs are also capable of data transfer overlapped with parallel/sequential instruction execution

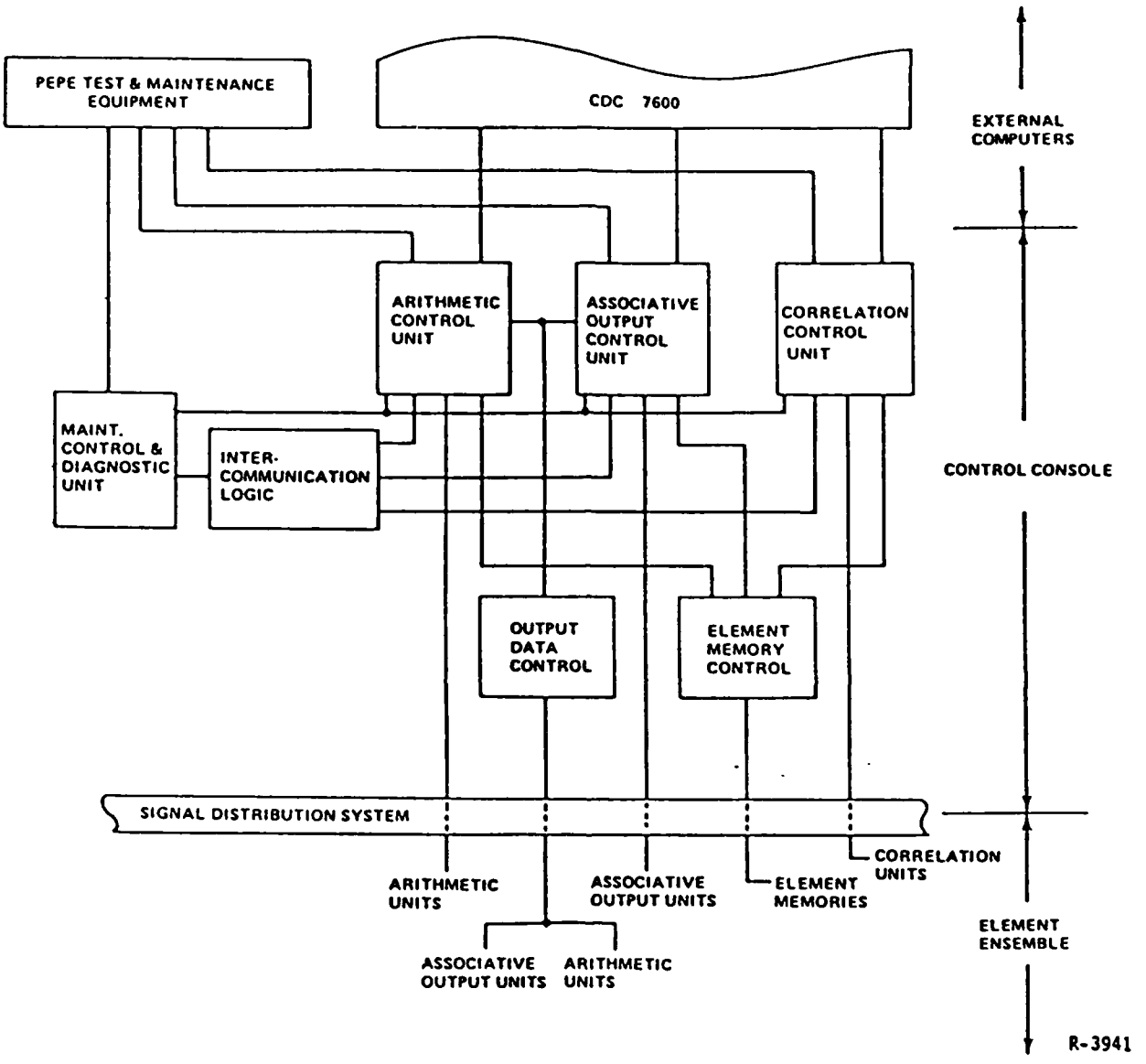



Figure 3-3. PEPE Control Console Components [21]

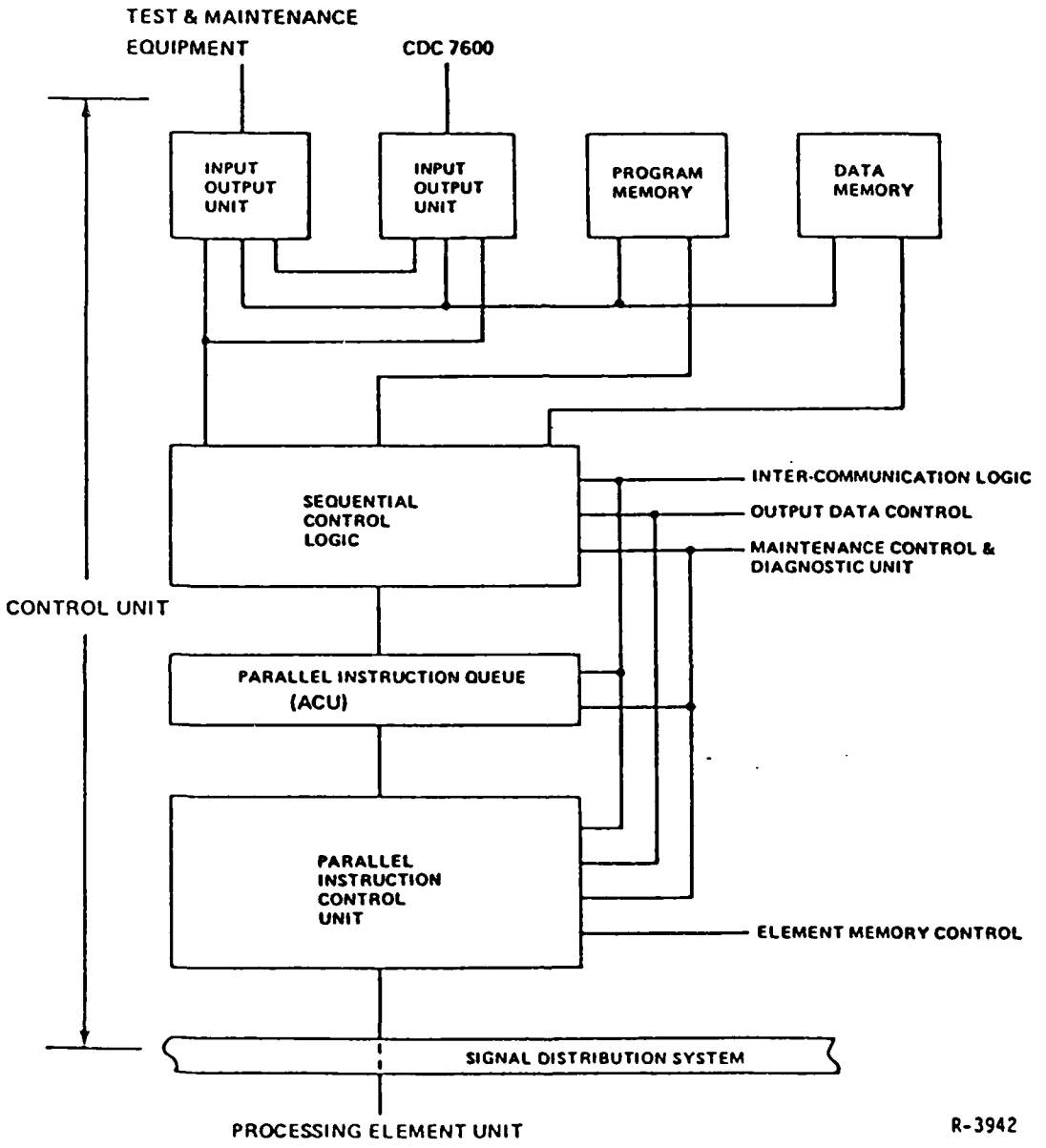



Figure 3-4. PEPE Control Unit [21]

PEPE is employed in BMD radar tracking by dedicating a PE and its memory to each target currently in track [25], [26]. Each PE stores within its EM the track state estimates (position and velocity), the new raw radar data to be used in updating the track, and requests for radar service. The PE memory is divided into identical fields so that the track data is stored in a homogeneous manner. This allows the tracks to be processed concurrently.

The important characteristic of the PEPE tracking algorithm is that it is a simple, range gating algorithm without any sophisticated enhancements. Due to the lack of time consuming complex computations, and the capability of parallel I/O, arithmetic, and associative operations, PEPE can exhibit very fast execution times along with high throughput.

Raw data is input to the PEs via the CCU. Range gates (upper and lower bounds on the acceptable measured radar range) are computed for each target in track and loaded into the corresponding comparison fields. The radar returns are then input sequentially by return, but in parallel over the targets. If the radar range falls within the limits of the range gate of any target the return is loaded into the raw data buffer of that PE. Figure 3-5 is a flowchart of a simple example of the correlation process.

After the data has been input the track estimates are updated. This update will only occur in PEs where new returns have been entered, all others are deactivated (masked). Since all PE memories are identical in form, the track update can be computed in parallel, with processing time independent of the number of tracks. After update the new predicted states are computed for all targets, and the cycle continues.

In order to exploit as much parallelism as possible in the tracking algorithm, data is accumulated for as long as possible. For instance, it is

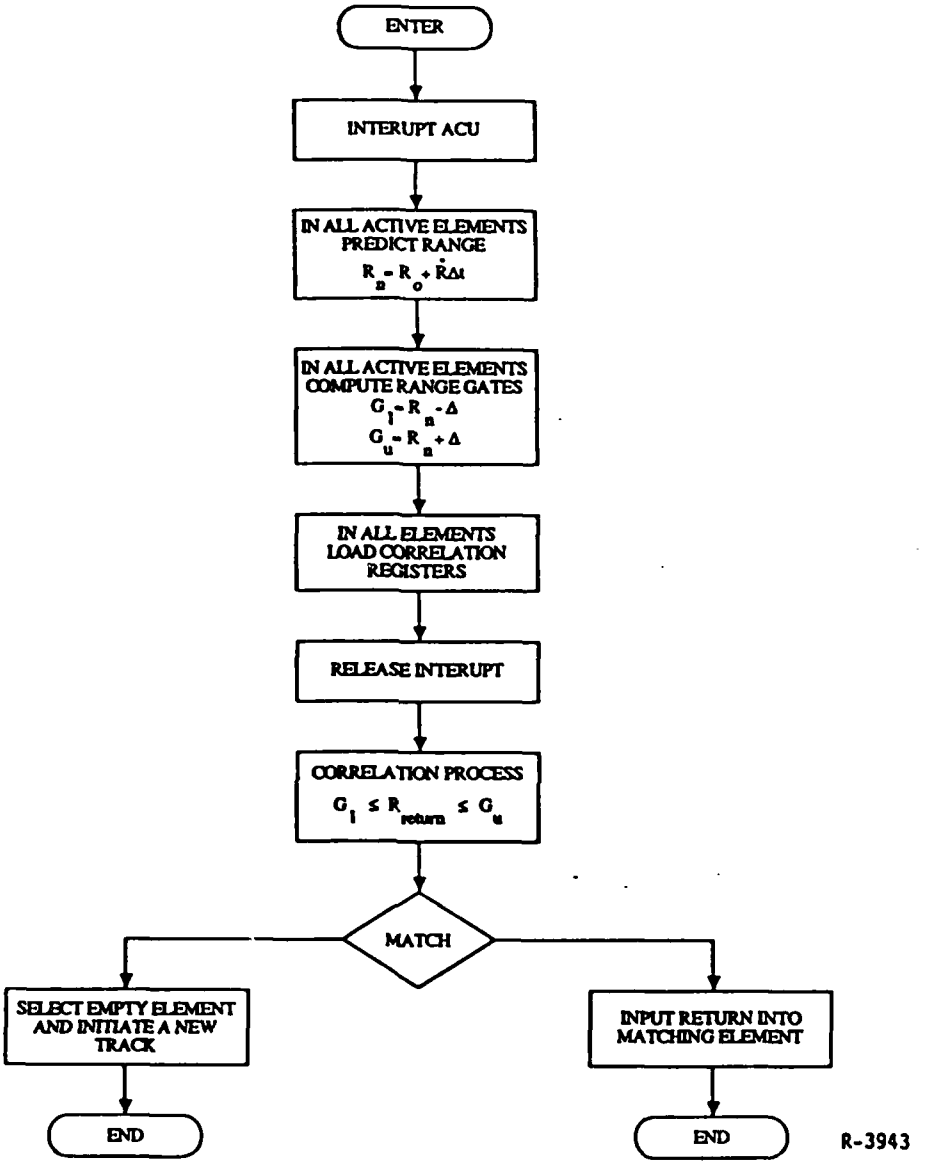



Figure 3-5. PEPE Correlation Process [25]

desirable to update all tracks concurrently, and to do this it is necessary that each track have been correlated with a return. If data is accumulated for too short a period, only a few of the tracks will have been correlated, and so PEPE will be operating with a degraded efficiency. Conversely, if the accumulating period is too long, processing on a track may not be completed before the next return for that track appears. The optimal solution is shown in Fig. 3-6. The accumulation interval is set equal to the period of time between observations on a single target minus the processing time [25].

From BMD benchmark tests run on PEPE, CDC 7700, and CRAY-1 involving multiple tasks (Kalman filter tracking routine and radar pulse scheduler) it was concluded that PEPE can outperform the other systems with as few as 108 elements [30]. It should be noted stressed that this conclusion was the result of a preliminary evaluation, and no further information has been encountered in the open literature.

Reference [26] discusses the method of implementing BMD data processing functional requirements on PEPE. These functions include more than just the target tracking discussed above, such as interceptor control and radar pulse allocation.

A similar algorithm has been proposed for air traffic control (ATC) using PEPE-like processors [27],[28]. A major difference with the BMD algorithm is that multiple targets are allocated to each PE in the ensemble. This allocation, based on spatial sectorization, is possible due to the predictable nature of the ATC problem. Also, real time display requirements preclude saving up scans of data, so the problem is less than totally parallel. Because of these reasons it is possible to assign targets to PEs such that no

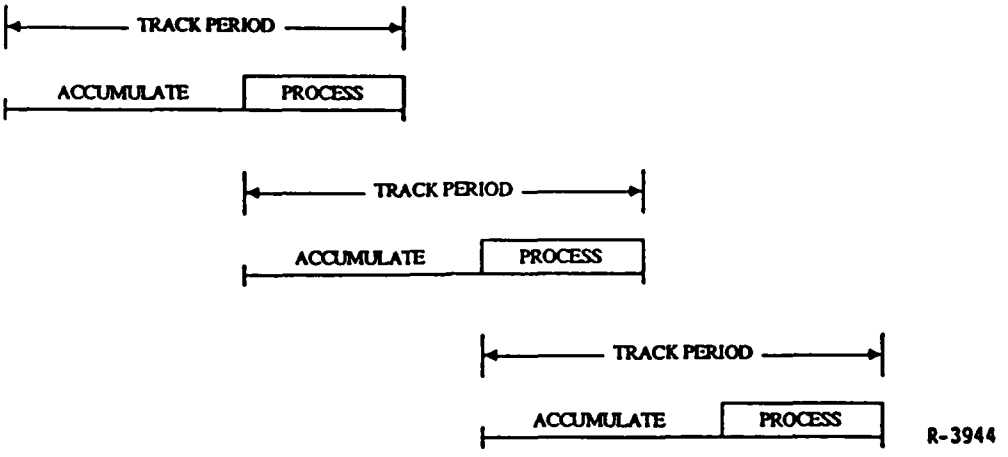



Figure 3-6. PEPE Parallel Process Scheduling [26]

two targets on the same PE will ever correlate with the same return, and so can be processed sequentially.

## 3.3 STARAN AND DERIVATIVES

The Goodyear Aerospace Corporation STARAN [15], [31] and its derivatives have been applied to both passive and active tracking, as well as air traffic control. STARAN is a bit serial AP containing up to 32 array modules. Each module contains 256 small PEs and a 256x256 bit Multi-dimensional Access (MDA) Memory. The PEs communicate with the MDA through a Parallel Input/Output (PIO) "flip" network which can permute a set of operands to allow inter-PE communication. The STARAN architecture has been discussed in detail in subsection 2.2.3 on associative processors, and so will be omitted here.

The principle multiobject tracking program employing STARAN found in the open literature is that of the Rome Air Development Center Associative

Processor (RADCAP) Project [32]-[37]. Specifically, this effort investigated applications of the RADCAP STARAN to USAF Airborne Warning and Control System (AWACS) Project data processing functions. The RADCAP test bed facility consists of a four-array module STARAN and various peripheral devices interfaced with a Honeywell Information Systems (HIS) 645 sequential computer.

The active radar tracking algorithm implemented on the RADCAP STARAN consists of two phases: association/correlation and Kalman filtering [34]. Association consists of selecting crude range and azimuth gates (i.e., minimum and maximum acceptable values) for each target in track and then testing the returns to see if they fall within the gates. The gate size is chosen from four sizes, depending on the correlation history of the track. Where multi-object tracking with PEPE performs this search in parallel over the targets (serially over returns) this implementation carries it out in parallel over the returns (serially over tracks). This is motivated by the large number of returns per scan relative to the number of targets, so that a higher order of parallelism is achievable by assigning one report to each PE. After the number of radar measurements have been reduced through gating the remainder of the calculations are performed in parallel over tracks. For the test cases reported in the reference the number of reports surviving the range and azimuth association tests was as low as two percent of the number of original reports.

Correlation, in this algorithm, is a more refined gating method based on track and measurement accuracies in order to reduce further the number of returns assigned to each target. Maneuvering targets are proposed to be handled by creating two copies of each track: one that choses reports based on a non-maneuver gate, and another that assumes that any report that failed the

non-maneuver gate is from a maneuvering target and does not employ a gate. These two tests can be performed in parallel. This allows a much greater degree of parallelism than first correlating non-maneuvering target returns, and then correlating the maneuvering targets with the remaining returns. The non-maneuver track is assumed correct until resolved on a subsequent scan.

It is still possible that more than one return has been correlated with a track, so a closeness criterion is employed to select the return to be used in updating. Kalman filter updating and prediction are then performed on all relevant tracks in parallel. The polar range and azimuth radar data is transformed to to Cartesian coordinates and two PEs are assigned to each track to exploit the decoupled x-y form of the Kalman filter equations, so that twice the parallelism can be achieved.

All of the tracking computation occurs in array module 0 (c.f. Fig. 2-7). Array module 1 contains the track history variables and module 2 holds intermediate variables which must be stored temporarily during association/correlation and filtering. Array module 3 serves two purposes. It acts as an I/O buffer for radar returns, transferred from bulk core memory by the PIO Control while Kalman filter calculations are being carried out in array module 0. The radar returns are then transferred in parallel to array 0 to start the next scan. Array module 3 is also employed as a backup store during association/correlation.

No timing results were available from the reference.

Automatic track initiation for active AWACS tracking on RADCAP is discussed in [35]. The basic concept is to initialize a track for any report that fails to be associated with an established track. If this new track is matched to a return in the next scan, it is upgraded to an established track;

if not, it is dropped. In addition to dropping newly created tracks, all tracks are examined to see if any report has been correlated with them in some period of time. This is accomplished by assigning a figure of merit (termed "firmness") to each established track. The range of firmness is from zero to seven, with zero used as a space holder and seven implying a track well supported by radar evidence.

Active tracking is accomplished using radars that measure range, range rate, and azimuth. Passive tracking employs only measurements of target azimuth information obtained from the radiation (i.e., jamming) source. AWACS passive tracking on RADCAP/STARAN is functionally similar to the active tracking discussed [36],[37]. Additional parallelism is exploited by not assigning tracks to PEs, but instead aligning the vectors (states, covariance columns, etc.) among the PEs so that calculations may be done in parallel across vector elements as well as across tracks. This is an attractive alternative as movements of consecutive blocks of data within the STARAN arrays is fast in comparison to floating point arithmetic functions. To accomplish this the AM is divided into 8 32 bit fields, and each field into 4 (the number of states) blocks. Each block contains 64 (the maximum number of tracks to be processed in parallel) 32 bit words. This structure allows, for instance, two 4-dimensional vectors for each target to be added in one step (excluding data movement) by suitable data alignment.

A STARAN derivative has also been proposed for application to the ATC problem [38], [39]. Dynamic data for a track is stored in a contiguous space within one word of the AM array. This data structure exploits the variable field length capability of STARAN by using reduced precision variables (e.g., 12 bits for position, 9 bits for velocity, etc.). The algorithm employed for

ATC includes a track quality based on the correlation results to determine when to drop target tracks. For a two-sensor, 750 aircraft traffic load, approximately 6.6 percent of the 4 array AP capability would be utilized.

### 3.4 THE AIRBORNE ASSOCIATIVE PROCESSOR (ASPRO)

Goodyear Aerospace, responsible for the production of both the Massively Parallel Processor and STARAN, developed an airborne bit-serial associative processor in 1978 [40]-[42]. The Airborne Associative Processor (ASPRO) has approximately the same organizational structure and processing capability as the three cabinet STARAN in a volume less than 0.5 cubic feet. ASPRO was designed for early warning radar surveillance and command and control applications aboard the NAVY E-2C Hawkeye aircraft.

The ASPRO architecture is shown in Fig. 3-7. The array unit contains over 2000 PEs and over 1 Mbyte of memory. Control signals for the array are generated within the array control. The register and arithmetic section controls the I/O of array data and generates array addresses. Program execution control executes the application program supplied by the control memory, and also drives the array control and register and arithmetic unit. The control unit contains the program memory and buffers data between ASPRO and the host processor.

The array unit consists of 17 array modules, each of which contains 128 PEs and four 32x4k bit arrays of multidimensional access storage. Only 16 of the array modules participate in an application; the seventeenth is a spare. Each 4k bit word functions as a working and storage memory for its own dedicated processor. Figure 3-8 shows the layout of the custom design VLSI PE/Flip network chip, four of which are contained in each array module. This

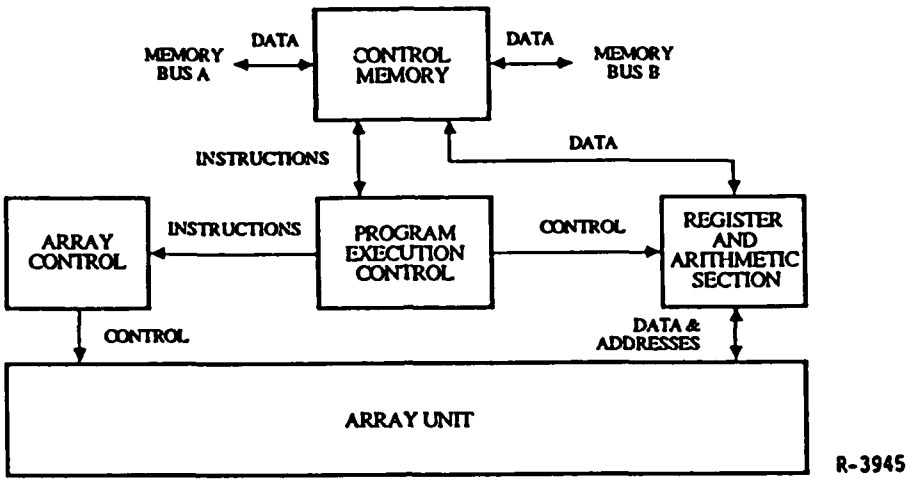



Figure 3-7. Block Diagram of the Airborne Associative Processor [40]

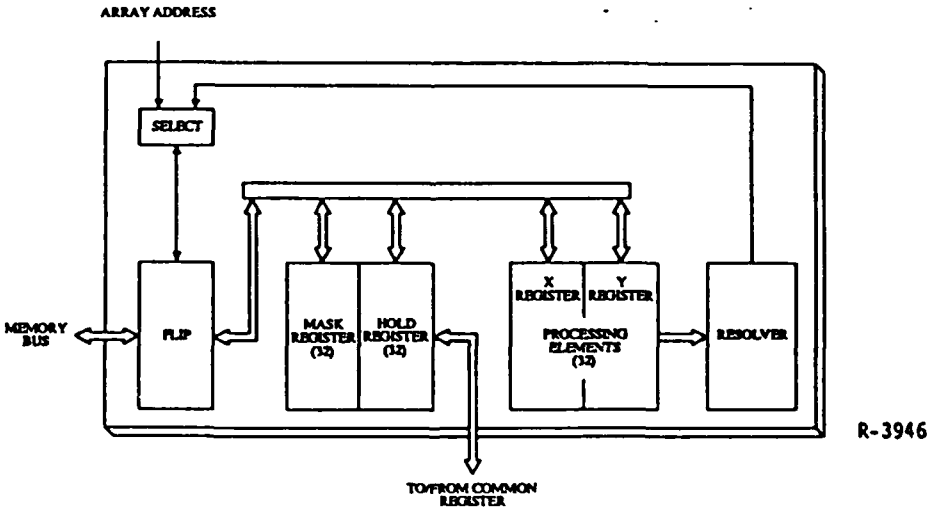



Figure 3-8. Custom PE/Flip Network Chip [40]

arrangement is very similar to that of STARAN, except that the flip network in the ASPRO is only 32 lines wide, instead of STARAN's 256. The MDA memory allows the register and arithmetic section to input or output a 32 bit operand in one memory cycle and the set of PEs to access one bit-slice from all operands in one memory cycle [40].

The PEs in ASPRO are almost identical to those in STARAN, with the exception of a one-bit hold register for masked write operations. High level operations are executed by the PEs on their array data by performing logical operations on bit slices of data read into the array registers, and then writing the results back into the array words. The set of operations includes the basic arithmetic operators (including floating point) and field compares for all the inequalities. The resolver on each PE/Flip network chip is used after associative searches to locate one (if any) PE whose data match that of the search.

The remaining units in ASPRO are similar in function to those of STARAN, and so will not be discussed further.

The airborne associative processor was designed to correlate radar reports from the E-2C with each other and with existing target tracks. Data is received by the E-2C from its own radar and from track data received over data links. This data is processed and correlated by the on board computers, and stored in track files. By selecting display options any of the three operators may call up a subset of the tracks. The display processing functions search the track database for the selected data, and output the appropriate data to the display buffers. No specific information on how the ASPRO is employed in multiobject tracking has been located in the open literature. It is not known, for example, what the division of processing between the

ASPRO and the host processor is. It would appear that one use of the ASPRO is in the parallel retrieval of selected track data, but other functions have not been specifically identified in the references.

An interesting related application of ASPRO as a parallel inference engine is described in [42]. Though it is not explicitly stated as such in the reference, it appears this work is focused towards designing a beyond visual range identification and threat assessment knowledge system. It is estimated that evaluating approximately 2000 rule antecedents in parallel will yield a throughput of over 1 million evaluations per second (this limit on rule antecedents is due to the number of words in ASPRO). ASPRO is employed as an inference engine by assigning a bit slice of the array to each state in the domain in the system. The setting of a particular bit in a word corresponds to that track (word) possessing that state. A rule is constructed by setting the individual bits that correspond to the facts that describe the rule. These rules can then be tested by iteratively loading 32 bit sections of a track and in parallel comparing it to the corresponding sections of the rules. Conclusions are carried out based on the results of the tests to update the database or perform some other instruction.

## 3.5 THE ASSOCIATIVE LINEAR ARRAY PROCESSOR (ALAP)

The Associative Linear Array Processor (ALAP) is a bit-serial processor developed by the Hughes Aircraft Company [43],[44]. ALAP is constructed from word cells, each containing arithmetic and control logic and one 64 bit word of shift-register memory. This is in contrast to computers such as STARAN, where the arithmetic and memory are separate. The ALAP cells are organized in a one-dimensional array (i.e., a line) as shown in Fig. 3-9. Connections to

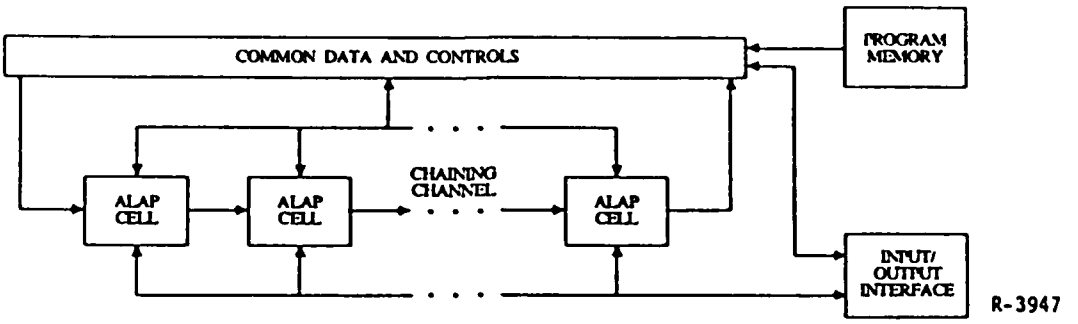



Figure 3-9. The ALAP Memory Array General Organization [43]

the cells are by means of three common data and control channels, and a chaining channel which provides communication between word cells in the line. All of these buses are bit serial. Two of the common channels permit data to be input to one or more of the cells simultaneously (data from more than one cell is logically ORed on the channel). The third common bus allows data to be output from one or more cells simultaneously. The chaining channel transfers data in one direction only from a cell to its nearest neighbor. The chaining channel can be used to sort data as it is being input to the array, to perform interword arithmetic, and other functions. A bi-directional chaining channel was considered for the ALAP but was discarded in order to keep the cost of producing an ALAP cell low. The first and last cells in the array may be left unlinked or linked together under software control to form a ring structure. All four communication channels are bit-serial in operation.

Figure 3-10 is a simplified schematic of the ALAP cell structure. Data storage for the cell is held in the 64 bit data register. This data register is used, along with data available over one of the common input channels (not shown in the figure) or the chaining channel, by the arithmetic logic. The state of the arithmetic logic is determined by the settings of the 6 bit flag

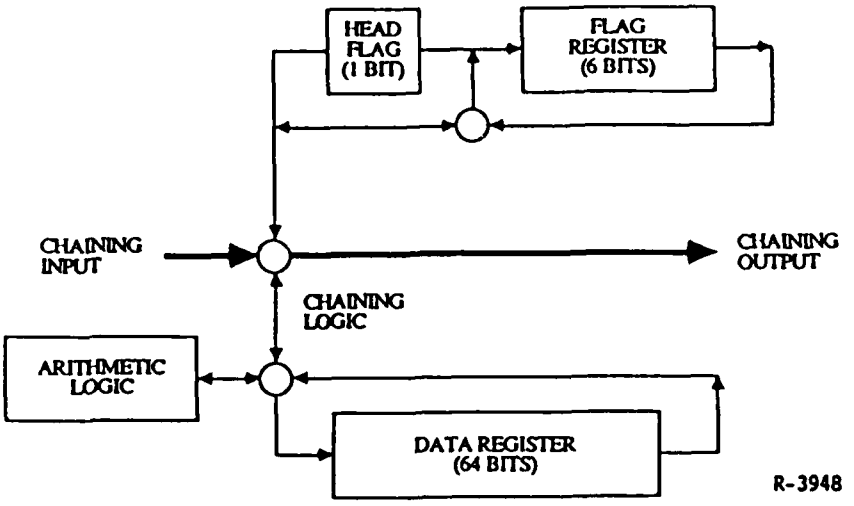



Figure 3-10. The ALAP Cell General Structure [44]

register. The 1 bit head flag allows the settings in the flag register to be rearranged as the data is shifted (interword flag data communication).

The ALAP memory is employed in instruction execution using the global mode called "Word Cycle." Word cycle operations consist of shifting the data registers in a selected subset of the cells, and either combining (arithmetically or logically) or replacing the register contents with the data on one of the three channels. The results of an arithmetic operation may replace the contents of the data register, or may be output on the chaining channel. Alternatively, the cell may act as a relay for chaining channel data, passing the incoming data on to the next cell. The cell control is specified through global commands, modified by the states of the local flag flip-flops. In this way different chaining channel actions may take place in different cells simultaneously.

The other two global modes are Flag Shift and Fault Isolation. Flag Shift is the set-up mode, used to set and transfer data among the head flag and flag register in each word. Fault Isolation is employed to remove bad words from the array.

The ALAP is programed for radar data processing in what is termed a "block-oriented" manner [44]. Blocks of contiguous ALAP cells are partitioned under software control, each block containing the data and working space for a single object under track. Within these blocks the track data is stored in a standard format so that similar data occupies the same relative location in all blocks. This is identical to the concept of fields in an associative array.

Track-While-Scan radar data processing on the ALAP is described in References [43] and [44]. Radar returns are assumed to contain range, range-rate, and the three direction cosines for each observation, with the observations grouped into scans (sets). The tracking functions are broken down to track correlation, association, and prediction. While it is not explicitly stated in the reference, it is obvious that the approach to tracking described is a fairly simple single hypothesis method using the closest return to the predicted observation value to update the track.

Track correlation is, in effect, a crude form of gating wherein the five track parameters are checked against preassigned ranges. All correlation calculations and data manipulation functions are performed parallel by track and serial by return. If all five observation values pass the correlation tests for a track, those values are loaded into the storage area of that track's block of cells (this storage area must be able to contain several radar returns). The correlation then continues with the next set of observation

parameters. Note that the correlation process may result in a given return being input into more than one track, and that a given track may have more than one return that passed its correlation test. Since these two results conflict with the implicit assumptions that a return can come from at most one true target, and that a target can produce at most one return, the process of track association is performed to eliminate both these cases of redundancy.

During the track association function, normalized error values are calculated for each track-observation pair that passed the correlation tests. Then, for those returns that correlated with more than one track, the track-observation pair with the smallest error value is kept and all others are deleted. This guarantees that each return has been associated with at most one track. It is still possible that some tracks may contain more than one return, and for those cases the return with the smallest error value is again retained. The first of these eliminations is carried out parallel by track and serial by observation, while the second is parallel by track and by observation.

The prediction function determines the expected values of the five track parameters. This prediction for each track will employ the newly associated measurement, if there is one. The algorithm used is a modified Kalman filter. All calculations are performed parallel by track. Additionally, parallelism within the prediction equations is also exploited. For example, there are a total of 47 multiplies for each track within the prediction algorithm, but they can all be evaluated in just four global multiply operations.

Some approximate timing information is given in the reference for the three tracking functions. Correlation requires a total of 22 + 24S word-cycle operations, where S is the number of returns in the scan. For an ALAP clock

rate of 5 MHz and 200 returns, the elapsed time for correlation is 8.3 msec (including 30 percent overhead). The computation of the normalized error value for all correlations requires a total of 146 word-cycles or 1.9 msec. The association process of forcing returns to be matched (at most) with a single track requires 9S + 5C word-cycles, where C is the total number of correlations for all of all the observations together. The elapsed time for 200 returns, 100 of which correlate to 2 tracks each, is 46.5 msec. The final association task of removing all the redundant returns for each track is done in parallel for all blocks, and requires 41 word-cycles, or 6.8 msec. Finally, predicting the track forward in time requires 475 word cycle operations, or 7.9 msec. It is estimated in Reference [43] that the crossover point between the ALAP (assuming a 10MHz clock) and a sequential computer (with an average instruction execution time of 1.2 5sec), above which the ALAP requires a longer execution time for the prediction function, is 16 tracks.

## 3.6 CONCLUDING REMARKS

In this section we have surveyed the significant existing parallel tracking methods and computers described in the open literature. Several important observations may be drawn from these systems. The most significant characteristic of all of these methods is that they employ rather simple single hypothesis algorithms to accomplish the multitarget tracking function. Another common trait is that the four tracking computers discussed (PEPE, STARAN, ASPRO, and ALAP) all exploit the content addressable memories in associative processors. This choice of APs is motivated by the ease in which the gating of sensor reports may be carried out by employing the parallel comparison capability of associative processors. Because of this historic concentration

on APs, we also will investigate their applicability to the specific tracking algorithm of interest here - that of track-oriented multiobject tracking, which generates multiple track hypotheses.

# SECTION 4

# OVERVIEW OF THE TRACK-ORIENTED MULTIOBJECT ALGORITHM

## 4.1 INTRODUCTION

A track-while-scan radar illuminates its region of coverage, sector-by-sector, by either mechanically or electronically steering its radar beam. The time required to illuminate the entire surveillance area, once over, is referred to as the scan time (or simply scan); the objective of the tracking algorithm is to reconstruct target tracks by correlating their radar returns from one scan to the next.

Target tracks may be reconstructed by correlating returns from all scans at one point in time, i.e., in a batch mode. For practical scenarios, involving several measurements per scan, such batch processing algorithms are difficult (if not impossible) to implement. Alternatively, recursive target tracking algorithms, which postulate and update target tracks using measurements in each scan, may be implemented for practical use.

The multiobject tracking algorithm under investigation here is a recursive, track-oriented, multiple hypothesis approach formulated within the framework of hybrid state estimation. The actual derivation and equations are of nominal interest here, as we are more concerned with the algorithm functionality and parallelism than its minutiae. Therefore only an overview will be presented; the reader is directed to the references for more detail [1]-[4].

## 4.2 OPTIMAL ALGORITHM FOR MULTIOBJECT TRACKING

Factors that make multiobject tracking a complex problem are:

1. Presence of false returns (clutter);
2. Probability of detection of targets is generally less than 1;
3. New targets may be initiated in any scan;
4. Old targets may be terminated in any scan;
5. Old targets do not follow exact straight line paths and may, in fact, execute maneuvers;
6. Measurements of targets are generally corrupted by noise.

All of these factors may be accounted for by formulating the problem within a mathematical framework which is termed Hybrid State Estimation. The general hybrid state model consists of continuous-valued states and discrete-valued states. Using measurements related to the hybrid state, it is possible to compute an optimal (minimum mean square or maximum a posteriori) estimate of the hybrid state. Variables in the multiobject tracking can be identified with the generic model as follows: the state (generally position and velocity) of all existing targets constitutes the continuous-valued state; the noisy range, angle, and range-rate measurements from targets and clutter at every scan constitutes the measurement; indicators for target status (straight line trajectory model, maneuver model) and measurement status (associated with target, false alarm) constitute the discrete-valued state.

The hybrid state approach indicates the form of the optimal solution for the multiobject problem; however, the postulation of all the possible values of the discrete state (referred to as global hypotheses) and computing their likelihoods poses a difficult combinatorial problem.

The track-oriented approach provides a systematic method for attacking this problem. Rather than representing global hypotheses in the form of a matrix (as done in [45] and [46]), this approach maintains a set of target trees and a list of target track (branch) combinations. The root of each target tree represents the birth of the target and the branches represent the various measurements it can be associated with in subsequent scans and the different dynamics that the target can assume. A trace of successive branches from a leaf to the root of the tree corresponds to a potential track of the target. The leaf of each trace is unique and is referred to as the track node of the target tree.

Each element of the global hypotheses list contains a set of pointers to track nodes. They represent the combination of tracks postulated by the global hypothesis which that element represents. By assumption, the collection of pointers in any one such global hypothesis cannot point to two track nodes that include a common measurement. This implies that there is at most one return per target per scan.

Global hypotheses can thus be constructed by first associating measurements with each existing (parent) track node and then forming combinations of the resulting tracks. The former step can, in turn, be split to account for different dynamic models for the target prior to associating it with measurements. This is illustrated in Fig. 4-1 where we have assumed that the target can either continue in its constant velocity trajectory (S) or be terminated (X); the constant velocity trajectory is next split to account for associations with measurements $r_1$ and $r_2$, and the possibility of a missed detection (D).

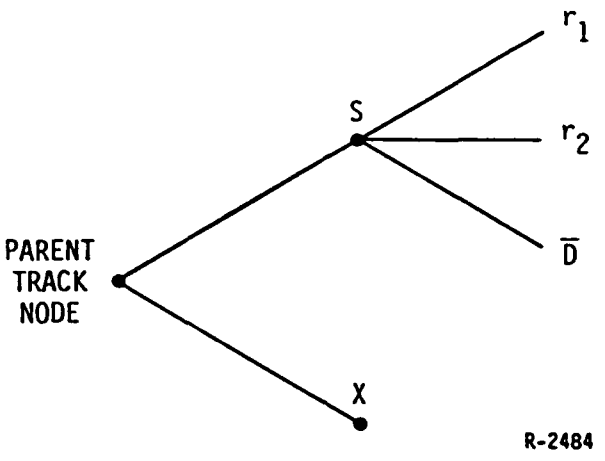



Fig 4-1. Track Splitting to Account for Different Target Dynamics and Different Measurement Associations

A typical representation of a set of global hypotheses is shown in Fig. 4-2. The dotted line indicates the points where new branches have been grown for existing target trees using the measurements in the most recent scan. Note that each global hypothesis can point to at most one branch of each target tree, and that the branches selected must have measurements that are mutually exclusive.

## 4.3 PRACTICAL ALGORITHM FOR MULTIOBJECT TRACKING

The track-oriented approach, discussed in the previous subsection, is a systematic methodology for constructing the optimal solution for multiobject tracking. However, for all practical scenarios which consist of several measurements in each scan, the computational requirements (both processing time and memory) of this algorithm will deplete the resources of any currently available computer. The reason for this problem is that the optimal algorithm postulates and retains all possible global hypotheses including ones that are only remotely probable.

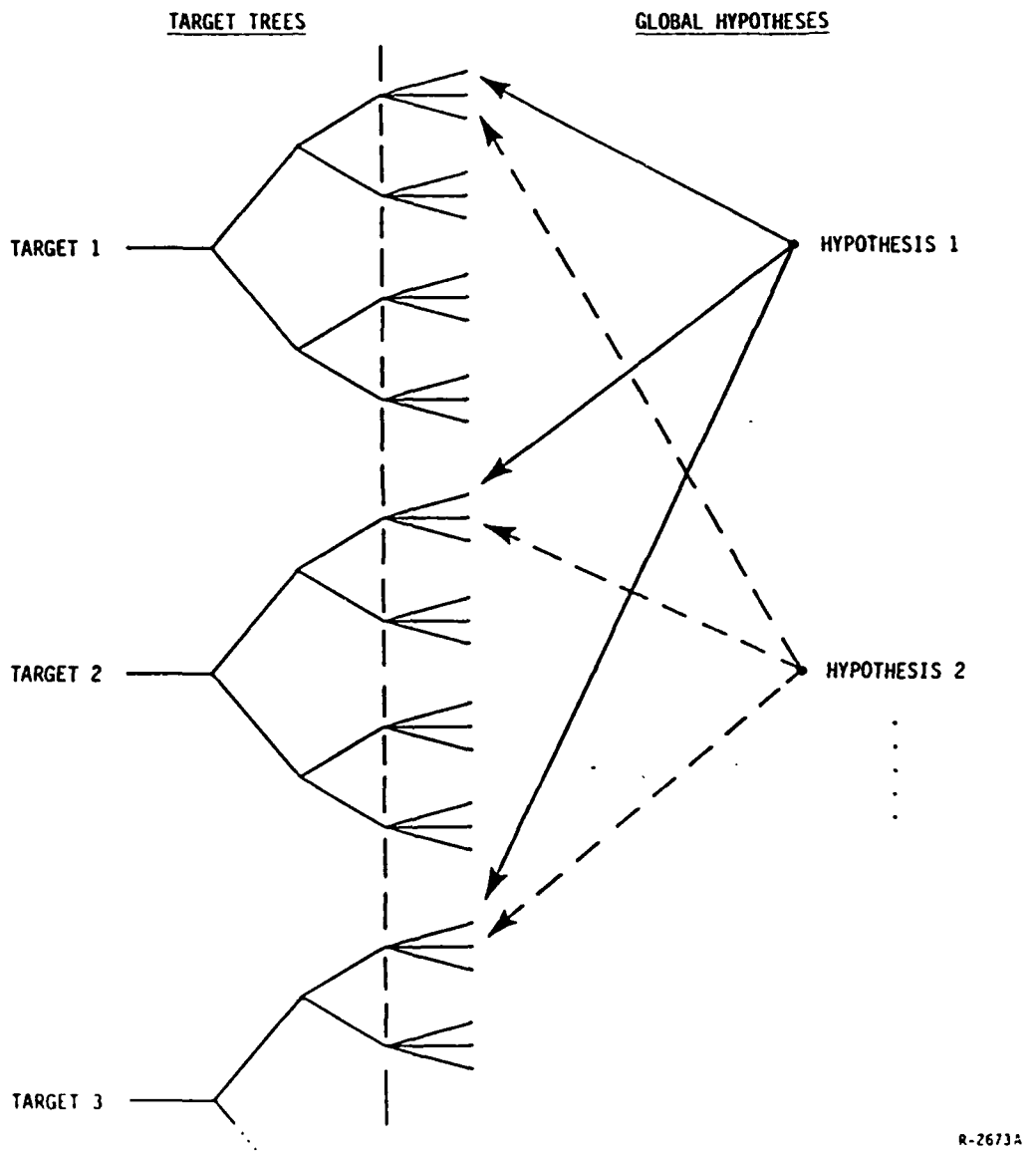



Figure 4-2. Representation of Global Hypotheses

In order to construct a practical algorithm, all such unlikely global hypotheses have to be eliminated. Since the track-oriented approach generates global hypotheses in two stages, such eliminations can be enforced at either the track generation stage or the global hypothesis generation stage. At either stage, eliminations can be achieved by screening (prior to generation of hypothesis) and/or pruning (after generation of hypothesis). The key techniques incorporated in the sequential form of the algorithm are discussed below.

Gating: Gating is a screening technique that eliminates unlikely associations of measurements with targets. It is very effective in cutting down the number of unlikely tracks and has been used in most tracking algorithms in the past. The gating process consists of constructing a region (gate) around a predicted target position, and allowing only those measurements which lie within this region to be associated with the target track.

N-Scan Approximation: In the track-oriented approach, target trees are 'branched-out' at each scan by associating measurements in the scan with existing branches in the target trees. The optimal multiobject tracking algorithm requires that each branch of the target tree be associated with each of the measurements in the scan, since all associations are possible. In reality, we know that each target should only have one branch corresponding either to an association with the measurement it generates or to no measurement in the case it was not detected.

Waiting for information in future scans to resolve measurement associations with th e current scan is particularly helpful in tracking crossing targets. An algorithm that waits N scans to resolve measurement associations in

the current scan is referred to as an N-scan algorithm. Note that the optimal algorithm will use an N equal to the number of scans for which tracking will be continued (i.e., N --> ∞).

The following steps summarize the N-scan algorithm:

1. Associate postulated targets with measurements for N scans into the future;
2. Form global hypotheses comprising such tracks with no measurement assignment ambiguities (disjoint);
3. Select a set which represents the most likely global hypotheses (in practice, only one is selected);
4. Drop target tracks not included in the most likely global hypothesis.

Classification of Targets: The process of gating measurements that we have discussed above is a form of screening that eliminates target tracks during the track generation stage. Another powerful screening technique that can be used at the global hypothesis generation stage is the selection of only a group of targets with which to form global hypotheses. This selection of targets is based on the criterion that the age of the target should be greater than a given age A. Targets that fulfill this criterion will be referred to as Confirmed targets.

The number of Confirmed targets will change from scan to scan since targets can be initiated and terminated at any scan. Since the set of Confirmed targets included in any global hypothesis at any scan should be a consistent set in that there exists no ambiguities in the assignment of measurements to targets, suitable rules should be formulated for promoting targets to the level of Confirmed targets. For this reason and also to allow a variable value of A, we have found it convenient to define three other groups of targets. The first of these, having an age exactly equal to A, is referred to as

Intermediate targets. It is from this group that Confirmed targets are selected. An Intermediate target is promoted to the status of a Confirmed target only if its presence does not cause any measurement assignment ambiguities with the existing targets.

Targets with age one are referred to as Born targets. Each of the measurements received at a particular scan could potentially represent the birth of a new target. If some of the measurements can be associated with Confirmed targets, we could rule out the possibility that these measurements represent new targets; however, since measurements are uniquely associated with a particular Confirmed target only after N scans (in an N-scan procedure), we assume that all measurements represent potential new targets.

The remaining targets, with ages between 2 and A-1, are referred to as Tentative targets. They represent a buffer group through which Born targets have to go through before they get promoted to Intermediate targets. Having these separate groups of targets makes it convenient for designing suitable data structures for the target trees. For example, Born targets have just one branch and one age group. Tentative targets could have several branches and several age groups. Intermediate targets could have several branches but only one age group. Confirmed targets should have a data structure similar to that of Intermediate targets so that Intermediate targets can be conveniently transferred to Confirmed targets. The different groups of targets and the order in which each is promoted to the next is depicted in Fig. 4-3. This form of grouping of targets by age is termed Classification.

Clustering: The computational burden in this multiobject tracking algorithm arises mainly during the formation of global hypotheses. The larger the

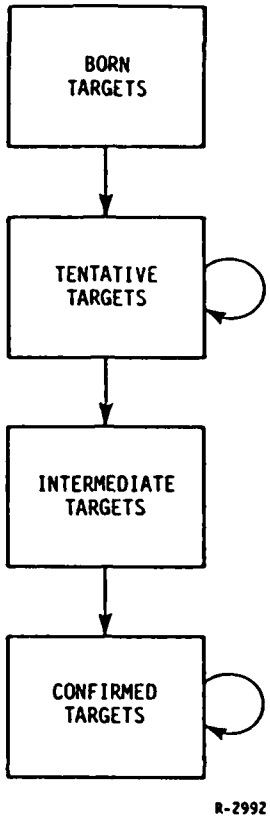



Figure 4-3. Classification of Targets

number of tracks that need to be considered, the larger the combinatorial problem. Classification of targets is one form of grouping that enables us to consider only a select number of target tracks while generating global hypotheses.

Since the purpose of forming global hypotheses is to resolve ambiguities in the assignment of measurements to targets, another form of grouping is possible. If targets lie in different regions on the surveillance area such that no common measurements are assigned to them, then obviously there is no need

to look for measurement assignment ambiguities among those targets. This motivates the need for grouping targets, based on geographical locations. We will refer to such grouping as Clustering. It can be viewed as a decomposition of targets into sets that enable processing of each group independently.

Clustering can be accomplished most readily by first forming subclusters during the association stage of the algorithm. For each scan collect the target numbers of all the tracks that accept a particular measurement. This will create a subcluster for each return in the scan that was associated with at least one target. These subclusters may then be combined to form connected clusters after all returns have been gated. The advantage to subclustering is that the number of subclusters will always be equal to or less than the number of measurements, and so reduce the combinatorial problem.

Figure 4-4 is a top level flow chart of the tracking algorithm. This level of of detail is sufficient for our needs here. In the following section we will examine the structure and computational requirements of the track-oriented approach in order to evaluate the effects of possible multiprocessor implementations.

## 4.4 CONCLUDING REMARKS

In this section we have briefly described the functional structure and justification of the track-oriented multiobject tracking algorithm. This algorithm is more involved than the single hypothesis methods examined in the previous section, and will therefore present a greater challenge to implement on parallel computers. Prior to such a step, a more detailed investigation of the computational size and structure of the track-oriented approach must be carried out. And that is the purpose of our next section.

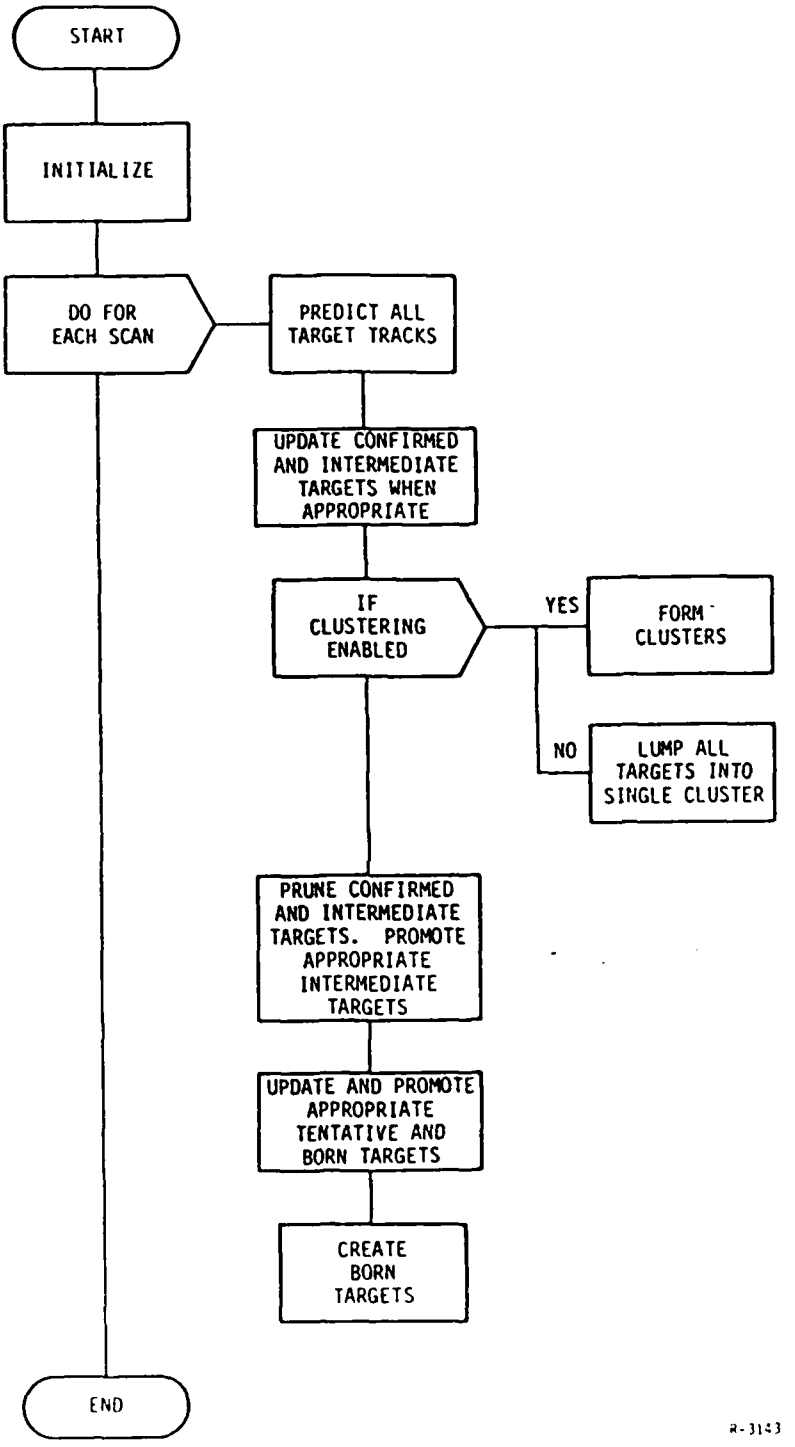



Figure 4-4. Top Level Flow Chart for Track-Oriented Multitarget Tracking Algorithm

# SECTION 5

# ANALYSIS OF THE TRACK-ORIENTED MULTIOBJECT TRACKING ALGORITHM

## 5.1 INTRODUCTION

An analysis of the track-oriented multiobject tracking algorithm is necessary prior to any discussion of possible implementation methods and computers. In the previous section we introduced the algorithm, and described its derivation and methodology. In addition to these issues, the structure and extent of the algorithm must be determined before an informed analysis can be made on appropriate computer architectures. It is the amount of exploitable parallelism, and the existence and location of sequential (non-parallel) computations, that must be clearly ascertained.

No effort will be made in this section to re-structure the computations to increase the available parallelism. Such efforts are best made when considering a specific computer architecture, and will be carried out in Sections 6 (associative processors) and 7 (MIMD computers). Determining the amount and type of parallelism, or independence, available in the algorithm itself allows us to choose the computer system that best matches the structure of the computations. The alternative approach would have been to make an a priori decision on the computer architecture to employ, and then "shoe-horn" the multitarget algorithm into it. Such an approach has obvious drawbacks, and precludes the choice of an "optimal" matching between algorithm and architecture. Also, by studying the algorithm first its theoretical performance on an idealized tailored machine may be investigated.

To accomplish this inquiry the algorithm must first be decomposed into its functional units, with an explanation of the nature and size of these units. The nature of the functional units, or tasks, may represent varying degrees of complexity, from simple calculations on operands (e.g., addition) to entire programs. In a bottom-up approach, the higher level tasks may be created by aggregating the more basic functions. Conversely, in a top-down approach greater resolution of the computation may be obtained by replacing a function with its component parts. In general, a task may be defined at any preferred level of abstraction. The decision on the degree of aggregation to consider in investigating the algorithm structure should be made based upon the possible classes of multiprocessor architectures to be eventually employed. For example, if multicomputer architectures alone (i.e., large grain parallelism) are under consideration, it is not necessary to examine the algorithm organization down to the operation level. Of course, if vector machines are under study then parallelism at the lowest level must be determined.

In addition to the tasks themselves, the dependency relations between individual tasks must be formulated. A dependency relation may be either a synchronization requirement (control) or a source/sink relationship (data), and therefore defines the precedence of the functional units. This ordering of the computational tasks is what defines their dependence or independence.

The size of the individual functional tasks is given by their computational requirements. The computational requirements of an algorithm may be specified in terms of operation counts and memory requirements. For a recursive algorithm such requirements computed for one iteration will be representative of the entire algorithm history, provided that the algorithm reaches some form of steady operation. Such a condition is certainly not achieved for

the optimal multitarget tracking algorithm since the computational requirements grow exponentially with every iteration. However, for the practical algorithm, which incorporates all the screening and pruning features discussed in Section 4, close to steady-state operation may be established.

In this section the computational structure of the algorithm will be examined, especially in regard to the parallelism inherent in it. This examination will be made independently of any implementation method or computer architecture. The operation count and memory requirements for one iteration (scan) of the tracking algorithm will also be presented. It will then be possible to calculate the maximum achievable parallelism and speed-up of the standard algorithm. Of course, there are other considerations and alterations that will affect a given multitarget algorithm implementation, and these will be covered in the subsequent sections.

## 5.2 FUNCTIONAL PARALLELISM WITHIN THE MULTIOBJECT TRACKING ALGORITHM

An algorithm generally exhibits parallelism at various levels of granularity at which operations can be defined. The instruction level can be thought of as the finest level of granularity. By aggregating operations at each level, coarser levels may be constructed. If the algorithm can be mapped on to a multiprocessor architecture that exploits the parallelism down to the finest level of granularity, then it would appear that the maximum possible speed-up can be achieved. Unfortunately, the finer the granularity of operations, the larger the overhead requirements. For example, if scaler multiplications and additions associated with a function evaluation are distributed over several nodes, then scheduling these operations at the different nodes and synchronization of the data on each arc could require substantial time

[48]. Clearly, there is an optimum choice of the granularity of operations at which parallelism in an algorithm should be identified.

We have chosen to investigate the parallelism of the practical track-oriented multiobject tracking algorithm at the procedure level. This is not to imply that we will only consider multiple computer (MIMD) architectures that process independent procedures concurrently. Both MIMD and SIMD computers will be considered for application in the subsequent sections. But we wish to avoid the involved analysis of the algorithm's minutiae that would be necessary to specify its complete parallelism. Below the procedure level, computations take the form of standard operations, e.g., matrix algebra. Parallel algorithms for such standard operations have been extensively studied by other researchers in the past (see [49] for an example of parallel algorithms for linear algebra).

The stated purpose of this research is to study the inherent parallelism of the tracking algorithm as an single entity, not merely to decompose it into a collection of random operations that may be implemented in a parallel fashion. Also, once the higher level parallelism has been identified, it may be possible to exploit the lower level parallel structure through the methods available in the literature. A final reason to investigate only the upper levels of parallelism is that the track-oriented multiobject tracking algorithm is known to possess an extremely high degree of inherent procedure level parallelism, and it is this type of parallelism that distinguishes the track-oriented approach from other tracking methods.

Parallelism in a computational algorithm can be analyzed and exploited only if the flow of data, dependencies among various tasks, and timing requirements during the execution of the algorithm are all clearly understood. The

best expository method found for displaying these characteristics is that of a computation graph [47]. A computation graph is a directed graph where the nodes represent some task (i.e., set of operations) in the algorithm, and the arcs represent dependency relations between source and sink pairs. This clear demarcation of computation and dependency is one of the attractive features of computation graphs. Another critical property of computation graphs is their effective hierarchical structuring of the parallel computations. Resolution of the graph to higher detail is obtained by replacing a node with a subgraph. Alternatively, more aggregated graphs can be created by combining subgraphs into nodes. These issues will become obvious in the subsequent discussion on the multitarget algorithm.

A computation graph, according to [47], can be thought of as "a program for some conceptual computation in terms of the operations of an abstract machine." In such an abstract machine there are a sufficient number of processors and communication channels to allow the program execution to be limited by its structure alone, and not by the computer architecture. The graph nodes are mapped onto processors and memories, while the arcs are mapped onto logical or physical channels. The required capability of these abstract processors and channels is determined by the level of aggregation of the original computational graph. We will make the assumption that all computations below the lowest level of the graph are done sequentially. This is done to specify a point at which the computation counts can be determined.

Figure 5-1 is a computational graph summarizing the steps executed in one cycle (scan) of the tracking algorithm. The nomenclature and functions included in the graph were defined previously in Section 4.

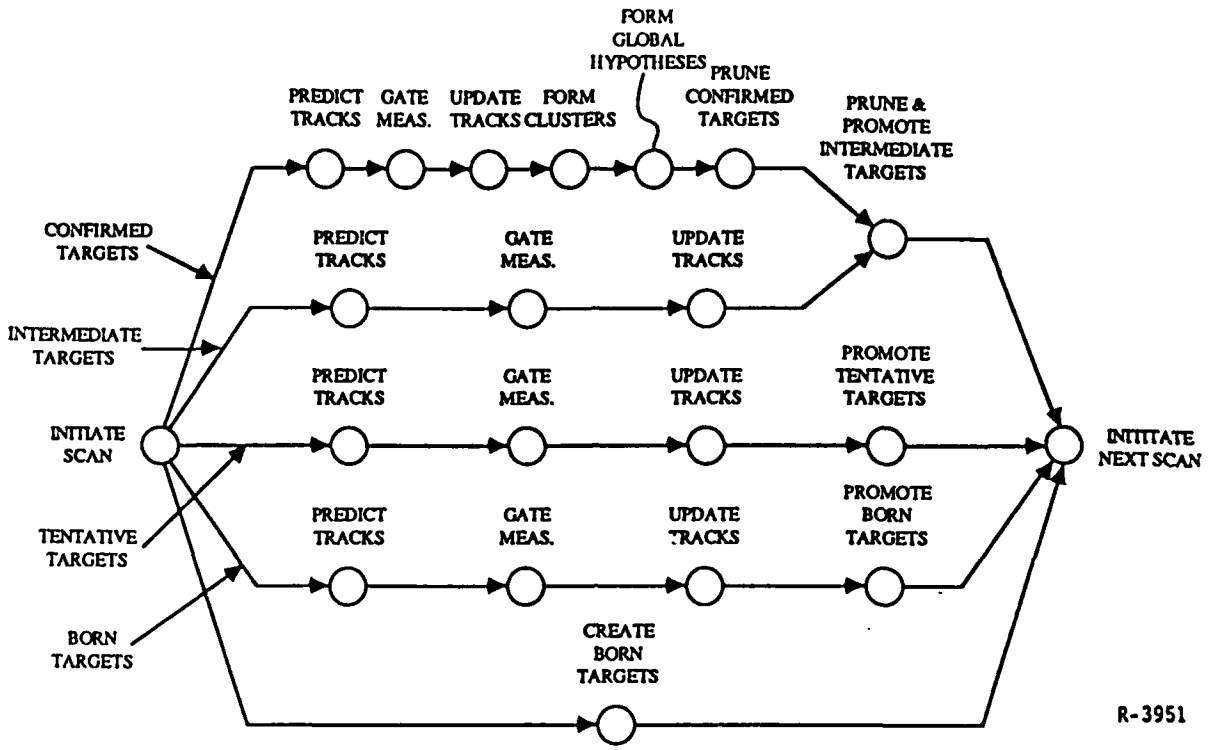



Figure 5-1. Computational Graph for One Scan of the Tracking Algorithm

Multiobject tracking algorithms are, in general, recursive algorithms, wherein the same sequence of operations is repeated for each scan of data. A computational graph for the entire algorithm history would merely be a chain of individual subgraphs, where each subgraph was identical to that of Fig. 5-1. A more compact representation of the entire algorithm would be to merge the INITIATE SCAN and INITIATE NEXT SCAN nodes into a single node. This would create a cyclical graph structure that matches the structure of the algorithm.

The processing for one scan of the tracking algorithm is initiated and terminated by the task labeled INITIATE SCAN (and the identically structured node INITIATE NEXT SCAN). This node represents two functions: the reception

of a scan of data from the sensors, and whatever processing is necessary to achieve the requisite tracking goals. The first function is obvious; a scan of data cannot be processed before that data is received. The second requires a more involved discussion.

The tracking operations summarized in the computation graph, while totally characterizing the multiobject tracking function, do not in themselves determine a tracking system mission. The target tracks constitute a database of information on the perceived state of the target environment, but no action has been specified to be taken on this information. Simply updating target tracks in computer memory accomplishes nothing of value. These estimated tracks must be output to a display, threat assessment function, targeting function, or some other mission-satisfying function. It is this requirement to output the target tracks that necessitates the terminal synchronization of the various functional paths in a single scan. In other words, all the processing for a single scan must be completed before the next scan is accepted by the algorithm. Were it not for this requirement there would be no reason to exclude the possibility of overlapping the processing for several scans of data. For example, there will always exist a sufficiently large enough scan of returns such that the processing of Confirmed targets will not complete before the next scan is received. The function of creating Born targets is a comparatively short task, and will typically have finished its processing. There is no algorithmic reason to not create Born targets in the new scan while the processing of Confirmed targets in the previous scan is still going on. It is the requirements of the higher level system functions, which require a complete report of the tracking algorithm's results, that synchronize the scan processing.

Once initiated, processing of the tracking algorithm proceeds along five primary paths during each scan. There is one path for each of the four target classifications (Confirmed, Intermediate, Tentative, and Born), while the fifth serves to create new Born targets from the returns in the current scan of sensor measurements. The independent structure (within a single scan) of each of the lower three paths is evident from the lack of common functional nodes. The upper two paths are not completely independent due to the common operation of PRUNE & PROMOTE INTERMEDIATE TARGETS. This almost total independence of the four target categories is an important characteristic of the track-oriented approach, due to its obvious potential for parallel processing.

At the hierarchical level of Fig. 5-1 the tasks shown refer to processing of entire classes of targets. While this is important to display the relationship of the top level functions, nothing in the graph addresses the processing of either individual target trees or target tracks. To address those concerns the paths may be replaced with a more detailed subgraph.

Figure 5-2 is a computational graph for the Confirmed target path alone. The Intermediate, Tentative, and Born target paths are all similar through the first three tasks. It is in Fig. 5-2 that the real opportunities for exploiting parallelism in the algorithm begin to become evident. Where the target class paths in the upper level computational graph are a series of sequential functional tasks, in the lower level graph they are shown clearly as a collection of functions displaying varied degrees of intrinsic parallelism.

The first three tasks in the Confirmed target path, PREDICT TRACK, GATE MEASUREMENTS, and UPDATE TRACK, are independent across all tracks of all the targets that were created in the previous scan. This independence is conditioned on the operations having access to the required measurement and track

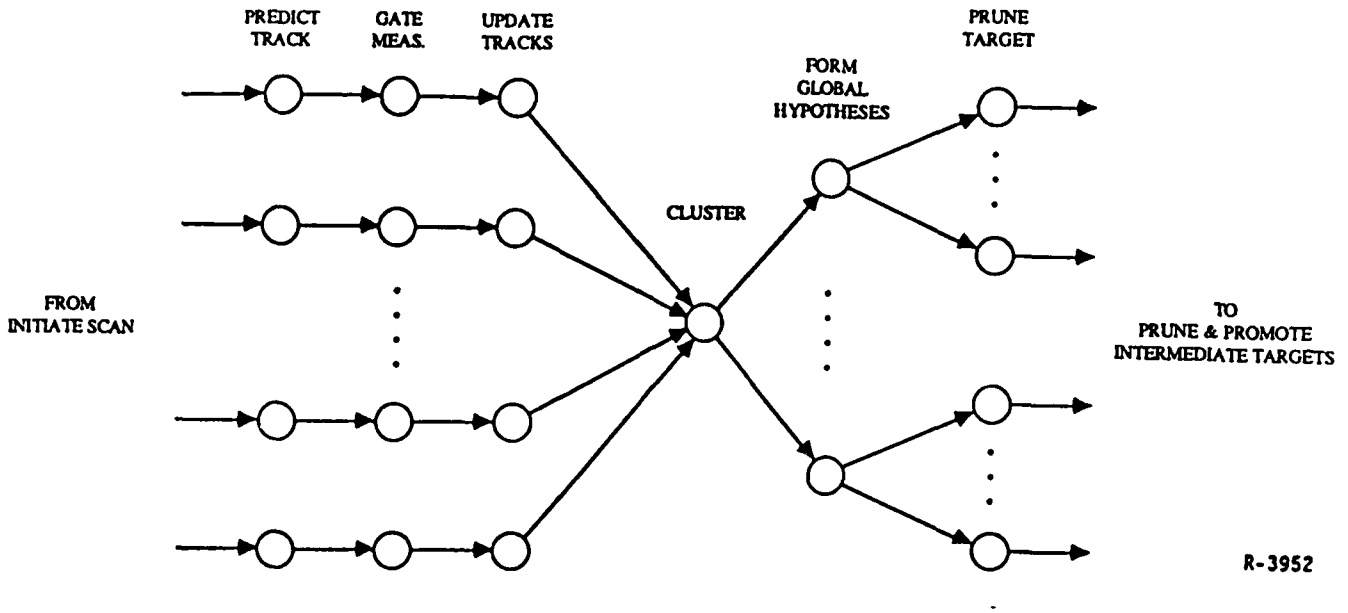



Figure 5-2. Detailed Computational Graph for Confirmed Target Path

data. In some computer architectures this may be difficult, unless there is either multiple copies of the data or simultaneous read access to the data. This concern will be deferred until the subsequent sections on specific implementation methods.

Beyond the independence of the track processing paths for these three tasks, they are also functionally identical for each path, only the track values are different. In the standard track-oriented algorithm under study here, all tracks are predicted ahead in the same manner, compared to the sensor returns in the same manner, and updated with the appropriate returns in the same manner. This identical processing has great potential for parallel implementations, both in multicomputers and in SIMD computers. As the three tasks possess identical computational structure they are perfect candidates for the kind of lock-step processing distinctive of array and associative processors. Of course, this is not a unique characteristic of the track-oriented approach.

All of the parallel tracking methods described in Section 3 exploited independence of the individual target tracks. But those tracking approaches were fairly simplistic and, in the nomenclature of Fig. 5-2, would terminate after UPDATE TRACK.

As we have stated, prediction, gating, and updating are common functions for all existing target tracks, and so exhibit parallelism at the track level. Additional functional parallelism exists below this track level for both the gating and updating tasks. Gating comprises two basic steps for each track: creating the gates and comparing the returns against these gates. The measurement gates themselves are functions of the existing track statistics and the measurement error statistics. Assuming that all returns in a single scan are of the same accuracy only one set of gates need be calculated for each track. All returns in the scan are then tested against this one set of gates. Therefore, the comparison operation has a functional level of parallelism equal to the number of returns in a single scan for each track. This gives a total degree of parallelism of the number of individual tracks multiplied by the number of returns in the scan. The computation graph for the gating function for a single target track is given in Fig. 5-3.

The comparison section of the gating procedure determines feasible track-return pairs for the current scan. Due to the multiple hypothesis nature of the track-oriented algorithm more than one return is allowed to be associated with each existing track. These new track-return pairs form the basis of the descendent branches in the target trees, and it is these track-return pairs that undergo updating. The level of parallelism in the update function is therefore equal to the number of newly created tracks, not the number of tracks that existed at the start of the scan.

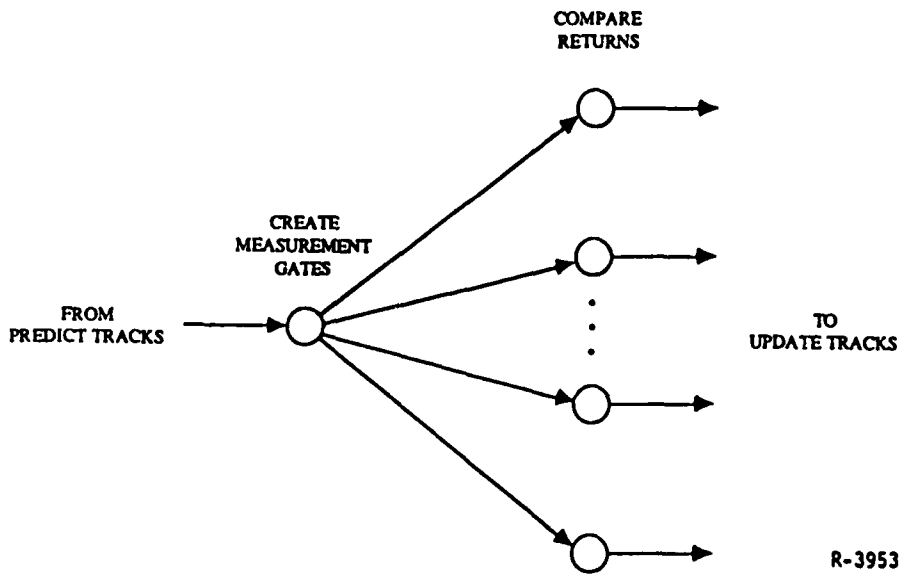



Figure 5-3. Detailed Computational Graph for Measurement Gating

Through the UPDATE TRACKS function, the computational graph of Fig. 5-2 is characteristic of all target categories, not merely Confirmed targets. The next three functions in Fig. 5-2 are exclusively Confirmed target operations.

The next functional task in Fig. 5-2 is CLUSTER. As discussed in Section 4, in clustering the target trees are grouped together in such a way that they form disjoint sets. Disjointness, in this context, requires that no two target trees from different clusters include a common return. There is a related requirement that any target tree in a cluster must have a common return with at least one other target tree in the cluster (unless the cluster is of size one). The simplest algorithm for creating clusters is to compare each target tree with every existing cluster; if there is a common return with a single cluster, the target tree is added to that cluster. If the target tree has returns common to several clusters, those clusters and the target tree are all merged into one new cluster. If there are no common returns with any cluster,

a new cluster is started. The clustering then continues with the next target tree. In practice this approach is modified somewhat to produce better execution times, but the principle is similar. Regardless of the implementation, an obvious characteristic of the clustering function is that it is not an independent, parallel task for each target or target track. This is evident in Fig. 5-2. The CLUSTER function creates a "bottleneck" in the computational graph, requiring that all preceding track level tasks have been completed prior to its initiation.

It is important to note that we are not maintaining that clustering must be processed completely sequentially. What we are stating is clustering, in general, is not independent across targets, and that its current algorithmic implementation is sequential. In the following two sections we will propose possible restructuring of the CLUSTER function to enhance its parallelism.

Once clustering of the Confirmed targets is completed, global hypotheses may be formed. The entire purpose of clustering the targets is to subdivide the global hypothesis formation task into a collection of independent smaller tasks. This is done because global hypothesis formation is basically a matching problem between target tracks, and so if all tracks were considered as one set, the computational burden would grow approximately combinatorially with the number of tracks. By partitioning the targets into independent clusters, independent global hypotheses can be formed for each cluster. The sum of the processing times for the individual clusters will be less than that of the full system, due to the replacement of the large combinatorial problem by several smaller combinatorial problems.

While clustering was originally introduced into the tracking algorithm to speed-up processing in a sequential computer, it also serves to produce a

level of parallelism in the global hypothesis function equal to the number of clusters (see Fig. 5-2). The global hypothesis formation within each cluster is sequential in the standard algorithm.

In the N-scan procedure adopted within the track-oriented algorithm after the most likely global hypothesis has been formed the measurement association N scans before is resolved. All Confirmed target tracks that do not include the resolved return are dropped, and then Confirmed targets with no tracks are dropped. All tracks for all Confirmed targets may be checked in parallel to determine if they contain the resolved return. A synchronization operation is then necessary for each target to "clean up" the data storage and to check to see if all tracks have been deleted.

This completes the discussion of the Confirmed target operations. The next task in the upper level computational graph (Fig. 5-1) is PRUNE & PROMOTE INTERMEDIATE TARGETS. Intermediate targets that use the same returns as those contained in the Confirmed targets, N scans earlier, are dropped. The remaining Intermediate targets are sorted and the most likely ones are promoted to the status of Confirmed targets. This promotion must succeed the PRUNE CONFIRMED TARGETS function as the number of Intermediate targets promoted depends on the amount of available space in the Confirmed target data structure.

In contrast to pruning of Confirmed targets, pruning of Intermediate targets may be carried out in parallel for each target, not for each track. This is because all Intermediate tracks in the same target tree have the same associated measurement N scans earlier, due to the age of the Intermediate targets. Therefore, only one track from each tree need be checked. The promotion of Intermediate targets is simply a movement of data between the Confirmed and Intermediate target structures. The specific method of accomplishing this

transfer is highly dependent on the implementation employed. For instance, in FORTRAN the data must be copied into new arrays, while in a language like PASCAL only the pointers need be altered. Since the new Confirmed targets typically must be numbered and inserted into the correct data location (either in an array or a list), promotion will be assumed to be a sequential function.

The functions PROMOTE TENTATIVE TARGETS and PROMOTE BORN TARGETS do not depend on the status of the respective higher target category's available memory. The oldest Tentative targets are updated into the Intermediate group, while all the Born targets are updated into the Tentative class. Both of these operations are simple data transfers as in the Intermediate target promotion.

The remaining function in the tracking algorithm computational graph is CREATE BORN TARGETS. Born targets are created for each return in the sensor scan by inserting the return values into target storage. No further processing is necessary. This function is obviously parallel over all returns, and is independent from all other target tasks.

This completes our discussion of the functional parallelism within the track-oriented multiobject algorithm. In the next subsections the computational requirements of each of the functional tasks will be analyzed.

## 5.3 DETERMINATION OF COMPUTATIONAL REQUIREMENTS

The screening and pruning features incorporated into the tracking algorithm will be referred to as algorithm parameters. These are chosen essentially on the basis of the anticipated scenario. For example, the number of Confirmed targets accommodated by the algorithm should correspond to the maximum number of targets anticipated within the surveillance region; the number

of tracks permitted for each target should take into account the clutter density, the proximity of other targets, and the probability of detection for targets. Similarly, the number of targets and tracks per target permitted for Intermediate, Tentative, and Born targets should be based on target birth and death distributions and clutter distribution.

Algorithm parameters provide a limiting influence on the computational requirements. However, the actual requirements in a particular scenario become a complex function of these algorithm parameters in addition to the scenario parameters (which is defined by parameters such as the statistical distribution of the targets and clutter). Rather than relating these requirements to these scenario parameters, and also accounting for the influence of the algorithm parameters, we have chosen to determine bounds on the requirements that are automatically imposed by the algorithm parameters. Specifically, data structures defined for storing target tracks and global hypotheses limit the computational requirements during an iteration of the algorithm. The resulting bounds allow us to ignore the effect of the scenario parameters in determining computational requirements. However, it should be noted that the choice of algorithm parameters has to be based on anticipated scenario parameters, and this choice forms a crucial step in the algorithm design.

Algorithm parameters bound both the operation counts and the memory requirements since they restrict the number of targets, the number of tracks per target, and the number of global hypotheses. From the discussion provided in Section 4, it can be seen that there are several such algorithm parameters. We will discuss the pertinent ones below.

Classification groups targets according to age. This grouping allows different data structures to be defined for targets belonging to different

groups. Confirmed targets, having the greatest age, are assigned a data structure which reflects the number of anticipated targets within the surveillance volume. The number of tracks permitted for each Confirmed target accounts for the associations with returns (correct or incorrect) received in each scan.

At the other extreme, Born targets, having the smallest age, are assigned a data structure which can accommodate all returns obtained in any scan as potential targets. Tentative and Intermediate targets are assigned data structures which allow the true targets to rise from Born to Confirmed category. The total number of tracks permitted may be summarized as follows:

$N_c$ - Number of Confirmed targets

$B_c$ - Number of tracks per Confirmed target

$N_i$ - Number of Intermediate targets

$B_i$ - Number of tracks per Intermediate target

$N_t$ - Number of Tentative targets

$B_t$ - Number of tracks per Tentative target

$N_b$ - Number of Born targets

$B_b$ - Number of tracks per Born target (=1 by definition)

In addition to maintaining tracks, the tracking algorithm also maintains global hypotheses. The number of global hypotheses that can be formed in any scan is an exponential function of the number and length of Confirmed target tracks; this number is limited by the data structures defined for storing global hypotheses and past history of the tracks. The maximum number of global hypotheses is limited to NGH and the stored history of each target track is limited to NSCANs in the past.

Clustering enables groups of Confirmed targets to be processed independently. The number of subclusters, NS, and the number of targets in each subcluster, NTS, limit the processing requirements for clustering. The number of Connected clusters (NC) and the number of targets per Connected cluster (NTC) limit the processing requirements for global hypotheses formation.

The flowcharts (Figs 5-1 and 5-2) identify steps of the algorithm which require major computational effort during each iteration or scan interval. We will estimate the computational requirements for each of these steps. For this analysis we will confine attention to floating point multiplications (divisions are treated as multiplications), floating point additions (subtractions and floating point comparisons are treated as additions), and integer number comparisons to determine the operation counts. Further, we confine attention to target track storage needs since it forms the major portion of the memory requirements.

Memory requirements for working space and program storage are not included due to their dependence on the specific implementation and coding method used.

The prediction step in Fig. 5-1 involves predicting tracks of all targets. The set of operations is identical for all target tracks and involves a Kalman prediction* (time update) which requires

$$\frac{3}{2}(n^3 + n^2) \text{ multiplications}$$

and

$$\frac{3}{2}(n^3 - n) \text{ additions}$$

*The Kalman filter can be implemented either in the normal form or the factorized form. The computational requirements of both are about the same [50] and the requirements specified here correspond to the normal Kalman filter.

where n is the number of states modeled in the Kalman filter. Storage space required for each track is

| | | |
|---|---|---|
| State | n | 32 bit words |
| Covariance | $\frac{n(n+1)}{2}$ | 32 bit words |
| Likelihood | 1 | 32 bit word |
| Measurement Indices | (NSCAN+1) | 16 bit words |
| Missed Detection Count | 1 | 4 bit word. |

Since the prediction step involves a 1:1 transformation for each track, target tracks computed during the prediction step may be stored in the old track location. Additional storage is thus not required during this step.

Update of each track can be perceived to have two stages. First, all returns are screened (gated) against the track and a fixed number of them are selected for association with the track. Next, the track is updated using the selected returns. Setting up the gate for each track requires

$m(n^2 + 2n)$ multiplications

$m(n^2 + n - 1)$ additions

where m is the number of measurements in each return. If $N_r$ denotes the number of returns per scan, gating of returns for each track (calculation of measurement residual squared and comparison to gate) requires

$mN_r$ multiplications

and $2mN_r$ additions.

The second stage of actually updating each track with each selected return represents a Kalman measurement update* requiring

$$m\left(\frac{3n^2}{2} + \frac{9n}{2} + 1\right) \text{ multiplications}$$

$$\text{and} \quad \frac{m}{2}(3n^2 + 5n) \quad \text{additions.}$$

Computation of the track likelihood involves

$$\begin{matrix} & 3m & \text{multiplications} \\ \text{and} & m & \text{additions.} \end{matrix}$$

Assuming each existing track gets associated with an average of R returns in each scan, the number of operations per track gets multiplied by this factor. Updating each existing track with R returns represents a 1:R transformation for storage requirements. Generally, R is greater than unity and so additional storage has to be provided during this step.

After updating of all tracks, the next step is to prune the unlikely ones. There is no pruning for Born and Tentative target tracks, i.e., all updated tracks in these groups get promoted to the next higher group. On the other hand, Intermediate targets are promoted only if it does not create a conflict in the most likely global hypothesis of retained Confirmed targets. Hence, this step has to await the completion of Confirmed target pruning. Further, if promotion of targets from one group to the next moves tracks into locations used by the higher group (to minimize storage requirements), then this has to be done sequentially.

*We have assumed that the measurements in each return are updated sequentially, i.e., as m scalar measurement updates as opposed to a vector update.

Pruning Confirmed targets involves the formation of global hypotheses and elimination of tracks not included in the most likely one. If target clustering is used, the formation of Connected clusters from Current clusters requires, at the most,

$$\frac{NTS\ (NS\ (NS-1))}{2}$$

16 bit word comparisons.

After the Connected clusters are formed, the formation of one global hypothesis requires*

$$\frac{NTC\ (NTC+1)(2NTC+1)\ NSCAN}{6}$$

16 bit comparisons and

$$(NTC-1)$$

multiplications. The upper bound on the number of global hypotheses per connected cluster is given by

$$\text{Min}\ \{\ (B_c + 1)^{NTC},\ NGH\}$$

so the total number of global hypotheses is the above multiplied by the number of connected clusters, NC.

Target tracks not included in the most likely glo l hypothesis are pruned away. The operation count for this stage of the pruning process requires only a few comparisons per track and we have chosen to neglect it.

*This is based on the assumption that it takes two tries to find an allowable combination for the second track added, three for the third, etc.

Based on the steady-state assumption, the pruning step represents an R:1 transformation in terms of storage requirements. Hence, we can expect the number of tracks per Confirmed target to reduce back to $B_c$.

Promotion of Intermediate targets is conditioned upon the available room in the Confirmed target data structure and can take place only after the pruning of Confirmed targets. Promotion of Tentative and Born targets to the next higher group is independent of the processing of the higher group. As mentioned earlier, storage requirements provided for each group dictates the sequence of operations. As in the case of Confirmed targets, the operation count for pruning and promotion of targets, and for the creation of Born targets is negligible and, for this preliminary analysis, we have chosen to ignore it.

Tables 5-1 and 5-2 are collections of the computational requirements given in this section. The first gives the computational requirements per track for the track operations (prediction, gating, and updating), while the second is the requirements for the Confirmed target functions of clustering and global hypothesis formation.

## 5.4 MEASURES OF PARALLELISM FOR THE TRACK-ORIENTED ALGORITHM

In order to provide a more concrete understanding of the relative computational loads of the various functional tasks a representative example will be given. We assume the following requirements for arithmetic operations:

| | |
|---|---|
| Time for 32 bit multiplication | 4 units |
| Time for 32 bit addition | 2.6 units |
| Time for 16 bit comparison | 1.3 units |

TABLE 5-1. COMPUTATIONAL REQUIREMENTS FOR TRACK OPERATIONS (PER TRACK)

| | MULTIPLICATIONS | ADDITIONS | INTEGER COMPARISONS |
|---|---|---|---|
| PREDICT | $\frac{3}{2}(n^3 + n^2)$ | $\frac{3}{2}(n^3 - n)$ | - |
| GATE (SETUP) | $m(n^2 + 2n)$ | $m(n^2 + n - 1)$ | - |
| GATE (COMPARISON) | $N_r * m$ | $N_r * 2m$ | - |
| UPDATE (STATES) | $m(\frac{3}{2}n^2 + \frac{9}{2}n + 1)$ | $\frac{1}{2}m(3n^2 + 5n)$ | - |
| UPDATE (LIKELIHOOD) | $3m$ | $m$ | - |

R-3949

TABLE 5-2. COMPUTATIONAL REQUIREMENTS FOR CONFIRMED TARGET OPERATIONS

| | MULTIPLICATIONS | ADDITIONS | INTEGER COMPARISONS |
|---|---|---|---|
| CLUSTER | - | - | $\frac{NTS[NS(NS-1)]}{2}$ |
| GLOBAL HYPOTHESIS | $NGC(NTC-1)$ | - | $NGC\left[\frac{NTC(NTC+1)(2NTC+1)}{6}\right]$ |

R-3950

Further, assume the following algorithm parameters for the multitarget algorithm:

$n = 4$
$m = 3$

$N_r = 100$

$N_c = 100$
$B_c = 6$
$R = 1.5$

$N_i = 20$
$B_i = 3$

$N_t = 30$
$B_t = 3$

$N_b = 100$

$NTS = 2$
$NS = 50$

$NTC = 4$
$NC = 25$

$NGH = 100$

The computational requirements for the various steps identified in Figure 5-1 may then be evaluated as:

| | | | |
|---|---|---|---|
| Predict Confirmed Tracks: | 600 * | 714 | time units |
| Predict Intermediate Tracks: | 60 * | 714 | time units |
| Predict Tentative Tracks: | 90 * | 714 | time units |
| Predict Born Tracks: | 100 * | 714 | time units |
| Gate Confirmed Tracks: | 600 * | 3,196.2 | time units |
| Gate Intermediate Tracks: | 60 * | 3,196.2 | time units |
| Gate Tentative Tracks: | 90 * | 3,196.2 | time units |
| Gate Born Tracks: | 100 * | 3,196.2 | time units |
| Update Confirmed Tracks: | 900 * | 825 | time units |
| Update Intermediate Tracks: | 60 * | 825 | time units |
| Update Tentative Tracks: | 60 * | 825 | time units |
| Update Born Tracks: | 90 * | 825 | time units |
| Clustering: | 1 * | 3,185 | time units |
| Global Hypotheses Generation: | 25 * | 16,380 | time units |

In order to define the relative magnitude of the processing loads, we will normalize the requirements for each step by the required time of the shortest step (that of Kalman prediction of one track). With this normalization, the computational requirements are shown in Figure 5-4. Processing each group of targets is shown in separate paths since they can be processed concurrently.

The operations of predicting, gating, and measurement updating are displayed as being carried out parallel by track, sequential by return. For the gating stage, it is conceivable that all returns can be screened concurrently for all tracks. Even for the next task of updating each track with selected

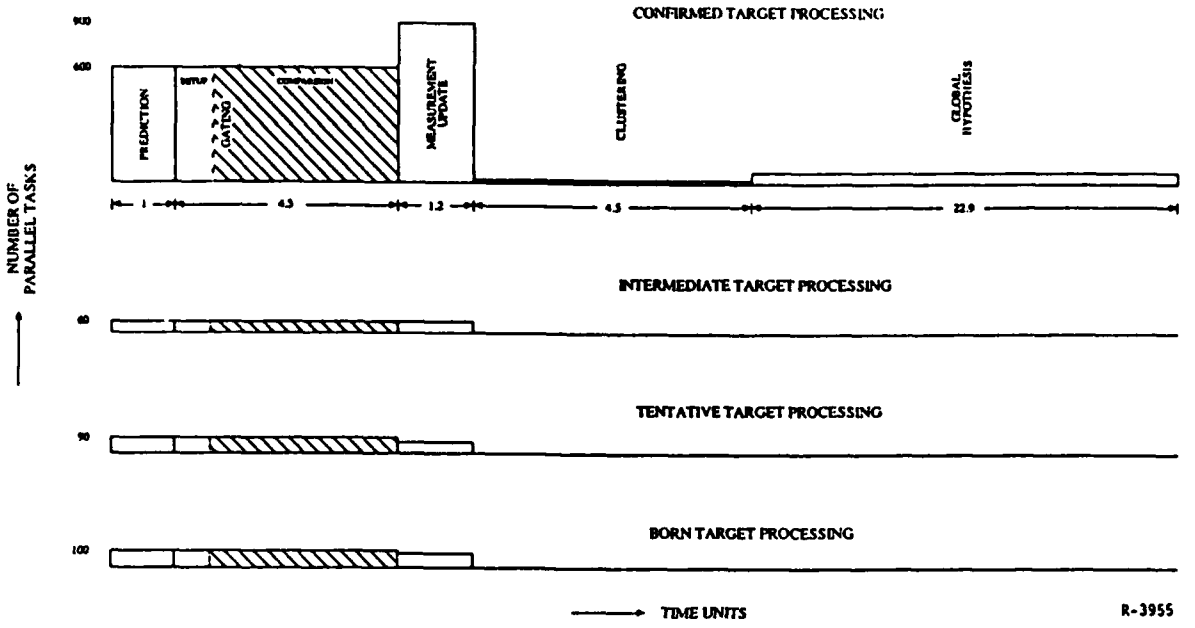



Figure 5-4. Representation of Parallelism in Tracking Algorithm

returns, it is possible that all such updates can be done concurrently. We have assumed a lower degree of parallelism due to practical considerations (to avoid requiring too many processors and copies of the necessary data).

The maximum number of tasks is contained in the path processing Confirmed target tracks. As the Confirmed target path is the critical path in terms of both the number of parallel processes and the required execution time, and as the functional tasks in the Intermediate, Tentative, and Born target paths are subsets of the tasks in the Confirmed target path, by concentrating on the Confirmed target path we will address the relevant issues in the track-oriented algorithm.

We assume the algorithm under investigation can be subdivided into N functional steps denoted $s_i$ (i = 1 to N), the $i^{th}$ of which can be further decomposed into $w_i$ parallel tasks. The functional steps take a time $t_i$ to

complete, assuming full parallelism. This is illustrated in Fig. 5-5 for the simple case of two functional steps. Obviously, these definitions are consistent with both the computational graph approach and Fig. 5-4.

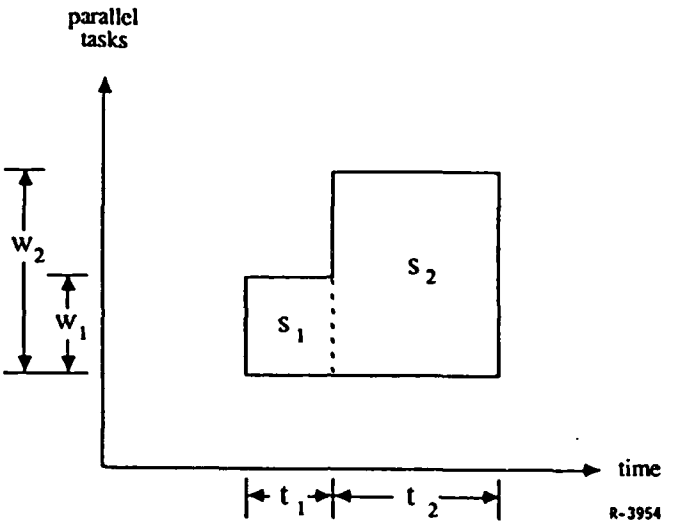



Figure 5-5. Graphical Display of Parallelism

For the Confirmed target path of the track-oriented algorithm the number of functional steps N is equal to 5, where:

| | | |
|---|---|---|
| $s_1$ | = | PREDICT TRACKS |
| $s_2$ | = | GATE MEASUREMENTS |
| $s_3$ | = | UPDATE TRACKS |
| $s_4$ | = | FORM CLUSTERS |
| $s_5$ | = | FORM GLOBAL HYPOTHESES |

The parallelism and processing requirements of the tasks are functions of the specific algorithm parameters mentioned previously. Using the values from the example of Fig. 5-4 gives:

| | | | | | | |
|---|---|---|---|---|---|---|
| $w_1$ | = | 600 | $t_1$ | = | 714 | time units |
| $w_2$ | = | 600 | $t_2$ | = | 3196.2 | time units |
| $w_3$ | = | 900 | $t_3$ | = | 825 | time units |
| $w_4$ | = | 1 | $t_4$ | = | 3185 | time units |
| $w_5$ | = | 25 | $t_5$ | = | 16380 | time units |

We define two ratios to characterize the relative size of the functional steps in the algorithm: the relative parallelism, $p_i$, and the relative computational requirement, $c_i$, where

$$p_i = \frac{w_i}{\text{Max } w_i} \qquad (5\text{-}1)$$

$$c_i = \frac{t_i}{\text{Max } t_i} \qquad (5\text{-}2)$$

The relative parallelism ratio is a measure of the comparative processor requirements of the individual steps. The relative computational requirement is a similar measure for the processing time.

For the given example, these measures are:

| | | | | | |
|---|---|---|---|---|---|
| $p_1$ | = | .67 | $c_1$ | = | .044 |
| $p_2$ | = | .67 | $c_2$ | = | .195 |
| $p_3$ | = | 1 | $c_3$ | = | .050 |
| $p_4$ | = | .001 | $c_4$ | = | .194 |
| $p_5$ | = | .028 | $c_5$ | = | 1 |

The measurement update stage is the most demanding in terms of the number of parallel tasks, while the clustering task has approximately one one-thousandth of the update task's requirement. With respect to the computation

times the global hypothesis task has by far the greatest requirements. These measures are not surprising given Fig. 5-4. The relative parallelism and computational requirements may be interpreted as the relative "height" and "width" of the individual steps. UPDATE TRACKS is the tallest step, while FORM GLOBAL HYPOTHESES is the widest.

A useful measure of the parallel nature of an algorithm is its parallelism ratio. We define a parallelism ratio $\rho$ as [29]:

$$\rho = \frac{\text{total (time * number parallel tasks) for each task}}{\text{maximum number of parallel tasks * total time}}$$

$$= \frac{\sum_{i=1}^{N} w_i t_i}{\underset{i}{\text{Max}}\,(w_i) \sum_{i=1}^{T} t_i} \qquad (5\text{-}3)$$

The parallelism ratio is a measure of the variation of the concurrency available within the different steps of the algorithm. For an algorithm with the same degree of parallelism at each step, $\rho = 1$. With the assumption that the number of processors in the abstract machine is equal to the maximum number of processors required, the parallelism ratio can also be interpreted as a measure of the utilized computer power. Continuing with the example of this section, the parallelism ratio is:

$$\rho = .16$$

The reason for such a low parallelism ratio is obvious from Fig. 5-4. The measurement update step displays a high level of parallelism (900 parallel tasks), while the clustering and global hypothesis steps have relatively low

levels of parallelism (1 and 25 parallel tasks, respectively). The low parallelism of the cluster and global hypothesis steps, along with their substantial processing requirements, "unbalance" the track-oriented algorithm. It is for this very reason that a considerable amount of effort will be spent in the next two sections to attempt to increase the parallel nature of these two tasks.

One of the advantages of the computational graph methodology is that it is readily extended to the concept of abstract machines. Most references discuss such topics as achievable speed-up and processor utilization with respect to given computer architectures. By assuming a sufficient number of processing elements and communication channels, and unrestricted access to all necessary data, these same concepts can be applied to the algorithm itself. The resulting measures will then represent characteristics of the tracking algorithm alone.

Mapping of an algorithm onto a multiprocessor architecture is equivalent to graph isomorphism provided that both the algorithm and the architecture are represented as graphs [51]. A graph for the tracking algorithm has been derived previously in this section. Rather than deriving the graphs for specific computer architectures, we assume that a flexible architecture that can be tailored to the requirements of the algorithm is available.

The speed-up of a fully parallel algorithm implementation over a purely sequential implementation (assuming that both the abstract and sequential machines have identical processor speeds) may be expressed as:

$$\eta_{max} = \frac{\text{time to process on abstract machine}}{\text{time to process on sequential computer}}$$

$$= \frac{\sum_{i=1}^{N} w_i t_i}{\sum_{i=1}^{T} t_i} \qquad (5\text{-}4)$$

For the above values the theoretical speed-up for the Confirmed target path is:

$$\eta_{max} = 144.1$$

That is, given our assumptions, an "optimal" parallel implementation of the Confirmed target path will operate 144.1 times faster than the same algorithm on a sequential computer.

Equation 5-4 has an underlying assumption that there is a sufficient number of processors available to fully exploit the inherent parallelism of the algorithm (= Max ($w_i$)). While this is certainly necessary to study the maximum achievable speed-up of the algorithm, it is also interesting to determine the speed-up when less than an optimal number of processors are involved. Whenever an individual algorithm function has fewer processors available than it has parallel tasks, those tasks must be partially processed sequentially. For instance, if a function has two parallel tasks but only one processor, the two tasks must be processed one after the other. The number of sequential partitions for a function with $w_i$ parallel tasks on P processors may be expressed as

$$\left\lceil \frac{w_i}{P} \right\rceil$$

where the notation $\lceil x \rceil$ is defined to be the unique integer satisfying

$$x \leq \lceil x \rceil < x + 1$$

The algorithm speed-up can now be given by

$$\eta = \frac{\sum_{i=1}^{N} w_i t_i}{\sum_{i=1}^{N} \left\lceil \frac{w_i}{P} \right\rceil t_i} \qquad (5\text{-}5)$$

For P greater than Max ($w_i$) the above speed-up equation reduces to that of Eq. 5-4. Figure 5-6 shows the Confirmed target path speed-up (for the example) on the abstract machine for values of P between 1 and 1000. Above P = 900 there is no further benefit in adding processors to the machine as the maximum level of parallelism in the algorithm itself is 900 parallel tasks. The algorithm exhibits almost linear speed-up at the lower end of the graph, but this falls off as more processors are added. It must be emphasized that

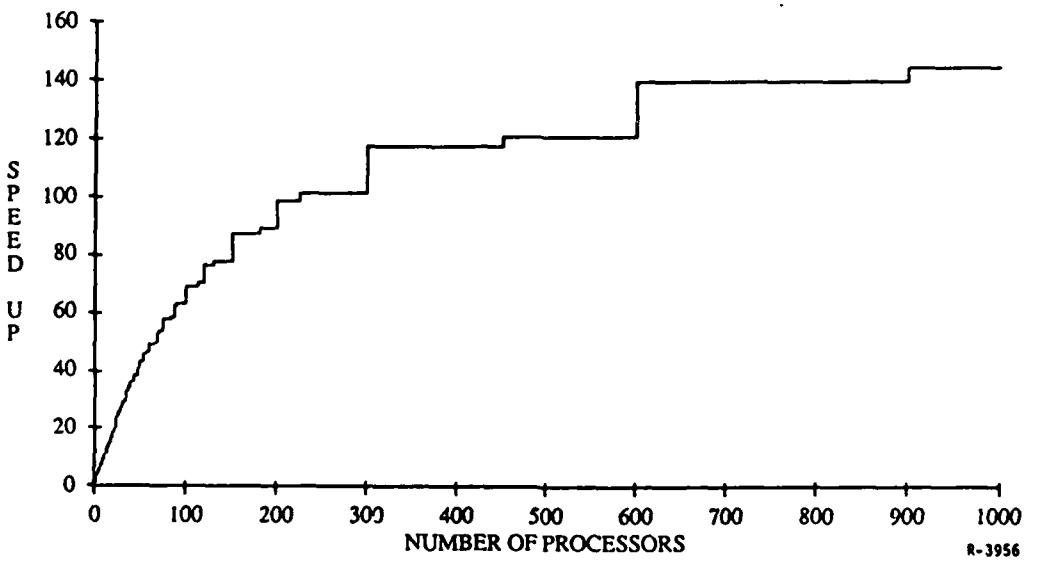



Figure 5-6. Speed-up of Confirmed Path on Abstract Machine

the speed-up displayed in this section only pertains to the abstract machine. The major omissions from this abstract model are those of communication constraints and memory access concerns. Instead of viewing these results as actual performance measures, they should be considered bounds on the performance of more realistic computer systems.

## 5.5 CONCLUDING REMARKS

As evident in the preceding discussions, the track-oriented multiobject tracking algorithm is far more computationally complex than the simple single hypothesis gating techniques previously employed on parallel computers (c.f. Section 3). The required number of calculations is far greater, as is the amount of synchronization functions (clustering and global hypothesis generation). The standard algorithm structure as described here displays varying levels of parallelism, from the track parallel functions of predicting, gating, and updating, to the totally sequential function of clustering. Without restructuring of the algorithm substantial processor inefficiencies will result due to the large amount of time required to perform the sequential tasks. This is not to say that the algorithm cannot be restructured. It can. How it can be successfully altered to exploit the available parallelism in an appropriate system will be shown in the subsequent sections.

# SECTION 6

# ADAPTATION OF THE TRACK-ORIENTED MULTIOBJECT TRACKING ALGORITHM TO ASSOCIATIVE PROCESSORS

## 6.1 INTRODUCTION

In the previous sections we have introduced both associative processors and the track-oriented multiobject tracking algorithm. We now combine the two. The selection of an associative processor (AP) as the initial multiprocessor architecture to investigate is motivated both by its historical application to tracking systems and, more importantly, by its pertinence to the given problem.

As was evident in the discussions in Section 3, most of the surveyed parallel tracking methodologies proposed or employed associative processors. These included the Goodyear Aerospace STARAN and ASPRO [15], [40], and the Parallel Element Processing Ensemble (PEPE) [21]. The inducements that drove these earlier applications are twofold: each object under track is processed in a lock-step, parallel manner, and all sensor reports may be compared (gated) with all target tracks simultaneously. These two generic tracking features imply an SIMD computer and an associative memory, respectively.

Another important feature of these simple algorithms is that, for the most part, they are single hypothesis approaches. Because there are no multiple hypotheses to appraise there is no need for such sequential, synchronization-type functions as clustering and global hypothesis formation. Therefore the entire tracking algorithm is functionally parallel by track.

Since there is an obvious distinction between the simpler tracking methods previously implemented on APs and the track-oriented algorithm, the relevance of APs to our approach may be of some question. The track-oriented approach contains many of the same functions that proved amenable to associative processing in the earlier algorithms - prediction, gating, and updating. Therefore, the only concern with the appropriateness of APs is how well they can handle the functions specific to the multi-hypothesis nature of the algorithm - track expansion, clustering, and global hypothesis formation. In the previous section we described these tasks and how they are currently implemented on sequential computers. The standard algorithms are completely sequential, and so should be restructured to exploit the capabilities of the AP. It is how successfully this restructuring is performed that will determine how appropriate associative processors are to the track-oriented approach.

In this section a generic associative processor architecture will be considered for application to the track-oriented multiobject tracking algorithm. The track-oriented multiobject tracking algorithm is well matched to APs, as it is a collection of fully parallel functions (predict, gate, and update) followed by search oriented functions (cluster and global hypothesis generation). It will be shown that the various tracking functions can all be implemented on the AP, each with a substantial degree of parallelism. The functions that display track level parallelism are the simplest to implement on APs. Clustering and global hypothesis formation are more involved as they must be restructured to exploit the specific characteristics of SIMD computers and associative memories.

## 6.2 TRACK-ORIENTED APPROACH TO ASSOCIATIVE TRACKING

The general approach of the associative track-oriented multiobject tracking algorithm is to operate on all appropriate tracks simultaneously whenever possible. If this is not possible, the computations should be restructured to take full advantage of the AP strengths. The primary advantage of APs over standard SIMD computers (e.g., array processors) is the capability to perform searches in parallel in each processing element. Since the functions of clustering and global hypothesis formation are essentially searches (one to form intersections, the other to determine disjointness) they appear to be well suited for APs.

The AM configuration of Fig. 2-6 will be used, along with the necessary control, memory, and I/O. The distinction between bit serial and bit parallel will not be made as it merely introduces a scaling factor into the operation counts (assuming uniform field widths). We will operate under the premise that communication between PEs (rows) should be kept to a minimum, and therefore all information for a single target track will be stored within one AM array word. This will preclude the exploitation of fine-grain parallelism (i.e., across vector elements) in the algorithm.

The assumed data structure required per target track (of all classifications) is given in Table 6-1, and illustrated in Fig. 6-1, where

| | | |
|---|---|---|
| n | = | number of states modeled/track |
| NR | = | number of returns/scan |
| NSCAN | = | number of scans in N-scan procedure |

This identical structure will be duplicated in each row of the AM that contains an active track. The rational behind and usage of these fields will be explained fully in the sequel. They are provided here so that the algorithm

TABLE 6-1. TRACK DATA STRUCTURE

| Field | Type | Number Required |
|---|---|---|
| States | Real | n |
| Covariance | Real | n(n + 1)/2 |
| Likelihood | Real | 1 |
| Target Number | Integer | 1 |
| Target Age | Integer | 1 |
| Cluster Number | Integer | 1 |
| Measurement Flags | Bit | NR(NSCAN + 1) |
| Update Flag | Bit | 1 |
| Active Track Flag | Bit | 1 |

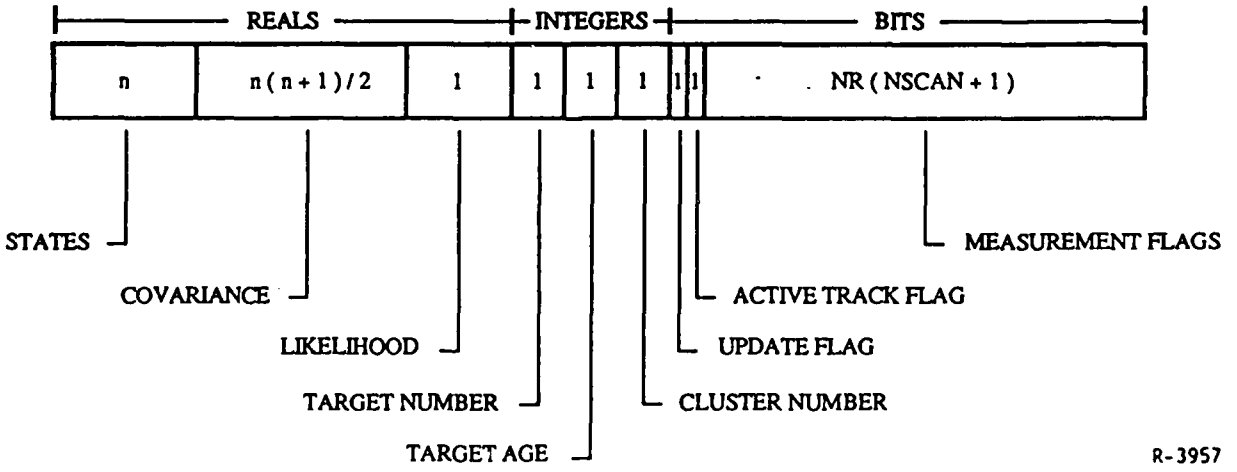



Figure 6-1. Associative Processor Track Data Structure

processing may be viewed in a unified manner. Beyond these specified fields, additional fields will be necessary per track for working space.

Only the upper triangular covariance elements are stored to reduce storage space. The target age is used to specify the track classification (Confirmed, Tentative, etc.) instead of using a bit flag for the individual types. The AP will then be able to select all tracks of a specific class with one associative search based on age.

The measurement flags are bit vectors, one bit for each return in the NSCAN + 1 (the additional scan is the current one) scans of interest. If a specific measurement is included in a track its corresponding bit flag is set to 1 and all other bit flags for that scan are set to 0. An alternative scheme would have been to insert the measurement numbers into (NSCAN + 1) integer fields, but this procedure would not allow the use of bit logic operations on the measurement sets. It should be noted that the storage necessary for bit flags will exceed that of the measurement indices for many realistic systems. The problem of weighing computational speed against memory requirements can only be evaluated for a specific system where their relative importance is known.

The update and active track flags are employed to signify that a track has accepted a return for updating and that the array word contains an active track, respectively.

## 6.3 ASSOCIATIVE PROCESSOR IMPLEMENTATIONS OF TRACKING FUNCTIONS

### 6.3.1 Prediction

Prediction (propagation) of states is a totally parallel process for each track, depending only on the local track information and the (assumed) common state model. The active track flag is set for all currently live tracks and

defines the words to participate in the prediction step. The operation count derived in Section 6 applies, except that here it is for all tracks, not just one. This Kalman prediction requires a total of

$$\frac{3}{2}(n^3 + n^2) \qquad \text{multiplies}$$

$$\frac{3}{2}(n^3 - n) \qquad \text{additions}$$

### 6.3.2 Gating and Track Expansion

Associative memories are well suited for gating, and it is probably for this reason that previous parallel trackers used APs. We will assume that all returns in a single scan have identical accuracies. This allows one gate to be computed for each track and then used for all measurements in the scan (this same assumption was made in Section 5). Setting up the upper and lower measurement bounds for all tracks requires

$$m(n^2 + 2n) \qquad \text{multiplies}$$

$$m(n^2 + n + 1) \qquad \text{additions}$$

$$1 \qquad \text{square root}$$

The square root was avoided in the previous section by gating on the square of the measurement residual, not the measurement itself. The computational penalty for using the square root is much less here, as it is computed for all tracks in parallel.

Once the gates are set up the actual comparisons can be made. Before gating, the update flags on all active tracks are set to 0, signifying that no measurement has been associated with that track. Next the active track flags are used to mask out all the empty words. This gating mask will remain constant throughout all measurement gating operations. The returns are loaded into the AM comparand register one at a time, and tested to see if they fall within the gate limits of each track in parallel. This requires

$$NR * 2m \qquad \text{associative searches}$$

to gate all returns in the scan.

After each measurement is gated it must be loaded into those tracks that satisfied the comparison test (the results of the test will be stored in the response register, which can then be used as a write mask). Two methods may be used to create new branches with the associated returns. The first is to buffer all measurements that are associated with a track into local word storage, and then create the new branches after the entire scan has been gated. The second method is to create new branches after each return has been tested. We will adopt this second method as it does not require a measurement buffer within each AM word. If a return has passed the gate test for a track, that track is copied into an open (not active) word along with the associated return, its update flag is set to 1, and the corresponding measurement bit flag is set to 1. The return is not inserted into the original track. This copy operation is best handled by an I/O control, as in STARAN [15], that can write an entire array word in parallel by bit, and in parallel with the AP control.

The gating then continues with the next return. Note that since the gating mask was set prior to any track creation, these new tracks will not participate in the gating. At the end of the scan there will be a collection of new tracks along with the original tracks which have no associated returns. These original tracks will be specified as missed detection extensions of the original tracks, and so will not be included in subsequent Kalman updating.

Born tracks are created after each return is gated by inserting the return into an empty array word, setting the measurement bit flag, and initializing the target number and age.

### 6.3.3 Updating

This operation occurs in three steps: process all active tracks with the update flag set to 1 and with age greater than 1 (Kalman update and likelihood); process all active tracks with the update flag set to 1 and with age equal to 1 (initialization); and process all active tracks with the update flag set to 0 (likelihood only). The operation counts in Table 5-1 may be used, giving

$$m\left(\frac{3n^2}{2} + \frac{9n}{2} + 1\right) \quad \text{multiplications}$$

$$\frac{1}{2}m(3n^2 + 5n) \quad \text{additions}$$

for Kalman update of all states for all tracks, and

$$3m \quad \text{multiplications}$$

$$m \quad \text{additions}$$

to update all track likelihoods.

### 6.3.4 Cluster Formation

Forming clusters differs from the previous functions in that it is not parallel over tracks. However, APs do offer advantages over RAM SIMD processors if the data is structured properly. In this subsection we will describe three approaches to associative clustering. The first employs the measurement bit flags introduced earlier. The second method presented attempts to avoid the large storage requirements of the bit flag approach by using the return indices (numbers) instead of bit flags. It will be shown that this method entails some prohibitive computational levels, and is provided in defense of and as a motivation to the bit flag approach. The final method described is motivated by an AM configuration with a limited number of bits per word. Due to this assumed memory constraint a rigorous clustering will be impossible; therefore a relaxed clustering tenet will be adopted.

BIT FLAG CLUSTERING

In order to exploit the unique capabilities of the AP, the data should be configured in such a way that the maximum number of returns may be processed in parallel. The concept of measurement bit flags was introduced in subsection 6.2. This approach is considered the optimum of the three to be presented, but may suffer from large data storage requirements in some scenarios.

Clustering occurs over Confirmed target trees, not on individual tracks. Therefore a temporary target data structure must be created. As with tracks, we assume one array word per target, thereby requiring NCT (= total number of Confirmed targets) words. The necessary fields per target are given in Table 6-2.

TABLE 6-2. TARGET DATA STRUCTURE FOR BIT FLAG CLUSTERING

| Field | Type | Number Required |
|---|---|---|
| Target Number | Integer | 1 |
| Cluster Number | Integer | 1 |
| Measurement Flags | Bit | NR(NSCAN + 1) |

The first step is to collect all the measurements included in each Confirmed target tree. This is done by activating all tracks for a particular Confirmed target, and then performing a parallel write of the track measurement flags (the communication channel is assumed to OR the flags, forming their union). This will require NCT parallel searches (compares) to create the confirmed target database. Next, the flags for the first target are loaded into the comparand register and a parallel associative logic bit-wise ANY* search is performed over all targets to see if any target has common measurements with the first. If there are any responders other than the original target, their measurement flags are combined and the process is repeated. The iterations stop when there are no new responders over the previous search. These targets are then labeled with the same cluster number, and masked out of all subsequent searches. The process is restarted with the next available target, until all targets have been included in a cluster. These cluster numbers are then copied back into the track words via NCT parallel searches.

*We will define an ANY search to be true if any two bits occupying the same position in the comparand and the field are both 1. This search does not appear to be standard, but should be simple to implement.

The maximum number of operations required to cluster the targets is:

3 * NCT associative searches

2*NCT searches are necessary to create the target data structure and to write the cluster results back into the track words. These searches are not a function of the actual cluster operations, and so do not vary from scenario to scenario (except with the number of Confirmed targets). During cluster formation, a worst case of NCT ANY searches occurs when there are no common returns between Confirmed target trees. This assumes that the Comparand register is large enough to hold an entire bit flag vector. If this is not the case then the vector will have to be partitioned into manageable fields. Processing the bit flags in sections will introduce a scaling factor into the operation counts for the clustering calculations.

A simple example of associative clustering is given in Fig. 6-2. In this example there are four Confirmed targets. The N-scan level is set at two, so three scans (NSCAN plus the current scan) of measurement flags must be retained. For this example there are three returns in each of the sensor scans. Target #1 has associated with it return #1 in the current scan, returns #1 and #3 in the previous scan, and no returns in the oldest scan. The other three targets are similarly configured. Prior to the first step in the clustering procedure the response and mask registers are cleared and two registers, either in the control system or the host, are initialized to one. The first of these registers will hold the number of the cluster being formed and the second will store the number of minimum number of search responders known to exist.

STEP 1.

| | |
|---|---|
| 1 0 0 : 1 0 1 : 0 0 0 | Comparand Register |
| 0 0 0 : 0 0 0 : 0 0 0 | Comparand Mask |

| TARGET # | CLUSTER # | MEASUREMENT FLAGS | M | R | T |
|---|---|---|---|---|---|
| 1 | 0 | 1 0 0 : 1 0 1 : 0 0 0 | 0 | 1 | |
| 2 | 0 | 0 0 1 : 1 0 0 : 1 0 0 | 0 | 1 | |
| 3 | 0 | 0 1 0 : 0 1 0 : 0 1 1 | 0 | 0 | |
| 4 | 0 | 0 0 1 : 0 0 0 : 1 0 0 | 0 | 0 | |

STEP 2, 3.

| | |
|---|---|
| 1 0 1 : 1 0 1 : 1 0 0 | Comparand Register |
| 0 0 0 : 0 0 0 : 0 0 0 | Comparand Mask |

| TARGET # | CLUSTER # | MEASUREMENT FLAGS | M | R | T |
|---|---|---|---|---|---|
| 1 | 0 | 1 0 0 : 1 0 1 : 0 0 0 | 0 | 1 | |
| 2 | 0 | 0 0 1 : 1 0 0 : 1 0 0 | 0 | 1 | |
| 3 | 0 | 0 1 0 : 0 1 0 : 0 1 1 | 0 | 0 | |
| 4 | 0 | 0 0 1 : 0 0 0 : 1 0 0 | 0 | 1 | |

STEP 4.

| | |
|---|---|
| 0 1 0 : 0 1 0 : 0 1 1 | Comparand Register |
| 0 0 0 : 0 0 0 : 0 0 0 | Comparand Mask |

| TARGET # | CLUSTER # | MEASUREMENT FLAGS | M | R | T |
|---|---|---|---|---|---|
| 1 | 1 | 1 0 0 : 1 0 1 : 0 0 0 | 1 | 0 | |
| 2 | 1 | 0 0 1 : 1 0 0 : 1 0 0 | 1 | 0 | |
| 3 | 0 | 0 1 0 : 0 1 0 : 0 1 1 | 0 | 1 | |
| 4 | 1 | 0 1 0 : 0 0 0 : 1 0 0 | 1 | 0 | |

R-3958

Figure 6-2. Associative Clustering Example

During Step 1 the measurement flags for the first Confirmed target are loaded into the Comparand register, and a parallel ANY search is performed in the measurement flag field for all words. Targets #1 and #2 both contain return #1 in the previous scan and so both respond to the search, as evident by the Response register. Since the number of responders has increased above one the response counter is updated to two and the processing continues for cluster #1.

In Step 2 the responders to the previous search (targets #1 and #2) write their measurement flags in parallel to the Comparand, forming the union of the flags. The other words are excluded from this operation by copying the Response register into the Mask register and then inverting it. Once the new comparand is formed, the Mask register is cleared, and the ANY search is repeated. Again a new responder is encountered as targets #2 and #4 have a common return in the oldest scan. The response counter is therefore incremented to three.

In Step 3 the first, second, and fourth targets' measurement flags are written in parallel to the comparand. Notice that this produces the same result as in Step 2 as Confirmed target #4 does not add any new returns to the cluster. After another ANY search there are again three responders, as expected. Since the number of responders matches the response counter the processing for the first cluster terminates. The Response register is then copied into the Mask register, inverted, and used to copy the cluster number from the global register into the appropriate array locations.

Since Confirmed targets #1, #2, and #4 have already been included in a cluster they will be masked out of all subsequent clustering operations. This step is not strictly necessary in forming the clusters as, due to the cluster

disjointness, any targets previously clustered will not respond to any new searches. It is necessary to maintain a list (i.e., register) of clustered targets to determine the next (if any) unclustered target with which to initialize a new cluster. For this example target #3 is unclustered, and so forms the basis of the second cluster.

In Step 4, the cluster counter is incremented to two and the response counter reset to one. The measurement flags of Confirmed target #3 are written to the Comparand and employed in an ANY search. There is only one response to the search, and so processing stops. The cluster number (2) is then copied into the Cluster field of the third target. Since there are no unclustered targets available to start a third cluster, the clustering function terminates.

The primary shortcoming in employing measurement flags for clustering is the large number of bits required. In many realistic scenarios with high clutter rates the number of returns per scan (NR) will be in the thousands. So, combined with a typical N-scan level (e.g., 3), a requirement of ten thousand bits per track is not unreasonable. The number of bits per word in STARAN and ASPRO are 256 and 4096, respectively, and so preclude the implementation of this method. It is expected that word lengths of 16k to 32k are imminent, with larger values still to come. Therefore, the measurement flag approach is considered feasible in the near future, if not for existing systems.

INDEX CLUSTERING

In the track data structure of Table 6-1 and Fig. 6-1, we assume that the number of the returns associated are stored in NR(NSCAN + 1) bit flags. This approach was not the first considered. An equivalent method to record the

returns associated with the track is to simply store the return indices (i.e., numbers) in NSCAN + 1 integer fields. Because the indices are stored as the binary representation of integers in integer fields, it will not be possible to operate on more than one return index at a time. This restriction precludes the highly parallel searches and set formations employed in the previous method. In the bit flag approach a given bit in a specified location could be interpreted independently of any other bits. This is not true in the present method. The smallest logical unit in the measurement index method is the integer field, or index.

As the returns are entered into the track database, they should be numbered such that there are no duplications within the last NSCAN + 1 scans. If the return indexing is reinitialized for each new scan, then both the scan number and the return number will be necessary to determine a specific return (e.g., return #5 in scan #2). A continuous numbering scheme will allow a single index to be employed. As only a limited number of scans are retained, and since only the current set of returns need have distinct measurement indices, the enumeration may be restarted periodically. This period will be at least

$$(\text{Max NR})(\text{NSCAN} + 1)$$

where (Max NR) signifies the absolute maximum number of returns that could ever be associated with any target in a single scan. This reinitialization should only take place between scans to avoid the computational penalty in testing to see if the period has been reached as each return is added.

If the AM supports variable width integers the required number of bits for all scans of interest is given by

$$\lceil \log_2[(\text{Max NR})(\text{NSCAN} + 1)] \rceil \; (\text{NSCAN} + 1) \text{ bits}$$

For all values of the term in brackets above 2, the number of bits necessary in this method will be less than the number required in the bit flag method. For example, if the maximum number of returns per scan is 1000 and the N-scan level is 3, 3000 bits will be required in the bit flag method, while only 36 bits are needed for the index method. If variable fields are not supported, then

$$\text{NSCAN} + 1 \qquad \text{integers}$$

are required.

In terms of track storage, there are obvious advantages to this approach. Unfortunately, the recording of associated returns is not an end in itself, but rather a means to facilitate the clustering of Confirmed targets, and eventually the formation of global hypotheses. We therefore must examine the possible clustering uses of the measurement indices. It is in this light that the current approach proves ineffectual.

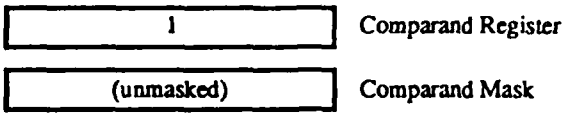
The first step in clustering is to create the Confirmed target data structure. Since the associated returns are represented as numbers, a list of return indices must be created for each target. These lists are merely the collection of all returns associated with some branch in the target tree. This can not be accomplished by forming the (bit) union of the individual target track fields, as was done previously. It can also not be accomplished by merely copying the values out of each of the target tracks in turn. If this were to be done it is quite likely that the same return index would appear in the target list several times. Therefore a method of deleting or avoiding duplications in the target list must be employed.

A possible method is to add the individual track indices to the target list one at a time, completing a scan for all tracks before advancing to the next scan. Prior to insertion in the list, all other tracks for the same target are tested to determine if they contain this same return. Those that do are masked out of any subsequent additions to the list for this scan. This will require one parallel search for every distinct return in the target tree, for each target. An undesirable result of this approach is that the list does not have an ordered structure. That is, the indices are shuffled. This may be overcome by searching for the minimum report index in all available tracks each time a new index is to be added.

An important characteristic of the Confirmed target's list of returns is that it could have as many as NR(NSCAN + 1) entries, corresponding to all the returns in all scans. This is highly unlikely, but it does point out the fact that the list does not have a predetermined length.

Figure 6-3 shows the data for the same example as in Fig. 6-2, except here the returns are represented using their indices. This example illustrates the primary difficulty in employing return indices - the index lists are "packed." Because the indices are appended to the lists as they are created, it is impossible to specify which field a specific index will occupy. Since the AP can only search over one AM array field at a time, it will be necessary to search over many fields to determine if a return is included in any of the target trees. In the example shown, index #7 occupies the third measurement index field in target #2, and the second index field in target #4.

The basic approach to index clustering is simply to form the aggregate of all targets having common indices. This is no different than the bit flag approach, but its implementation is constrained to look for common list




| TARGET # | CLUSTER # | MEASUREMENT INDICES | | | | M | R | T |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 4 | 6 | | 0 | 1 | 0 |
| 2 | 0 | 3 | 4 | 7 | | 0 | 0 | 0 |
| 3 | 0 | 2 | 5 | 8 | 9 | 0 | 0 | 0 |
| 4 | 0 | 3 | 7 | | | 0 | 0 | 0 |

R-3959

Figure 6-3. Associative Index Clustering

elements one index at a time. That is, return indices will be loaded into the comparand one at a time, and all targets will be searched to determine if they contain the identical index. Because the array fields must be processed sequentially, it behooves us to search as few fields as possible. If the measurement indices had been randomly ordered in each target list, then all fields of each target would have to be examined. As the indices are ordered, it may be possible to reduce the number of feasible locations in which to look.

Let

$v_\ell$ = index to be compared,

$1 \leq \ell \leq \underline{N}$

$f_{ij}$ = measurement field j of target i,

$1 \leq i \leq NCT$

$e_{ij}$ = index in $f_{ij}$

$S_i$ = set of indices for target i,

$N_i$ = cardinality of set $S_i$

There are total of $\underline{N}$ distinct indices in all targets, with the value $v_j$ corresponding to the measurement index. While it is not necessarily true that

$$v_\ell = \ell$$

as in our example, we will operate under the assumption that it is the case. This requires a renumeration of the indices.

We will make the second assumption that all indices in set $S_i$ are ordered within the set, i.e.,

If

$v_m, v_n \subset S_i,$ and

$e_{ij} = v_m, \quad e_{ik} = v_n,$ then

$j < k \iff m < n$

We wish to compare all the elements of one word, or set, to all other words in order to determine the intersection of a target index list with all other targets. This would appear at first glance to be trivial on an AP, but it is not. Complications arise due to the "packing" of the words. Because of the packing field j (of any word) will, in general, not contain value j. So if we are attempting to find all matches with the value found in a particular field of word i we must search in other fields as well.

An important assumption made at this point is that the value of the index that is being searched for is not known, but the field in which it was found is known. The relevant question is "In what fields $f_{kl}$ must you search to find a value identical to the one in field $f_{ij}$?"

It can be shown that

$$f_{ij} \quad \text{may contain} \quad v_j \text{ to } v_{[j + (\underline{N}-N_i)]}$$

and

$$v_j \quad \text{may be in} \quad f_{i,\text{Max}\{j - (\underline{N}-N_i),1\}} \text{ to } f_{i,\text{Min}\{j,\ N_i\}}$$

The minimum value of $e_{ij}$ is $v_j$. The lowest field of word k this value may be in is

$$f_{k,\text{Max}\{j - (\underline{N}-N_k),1\}}$$

The maximum value of $e_{ij}$ is $v_{[j + (\underline{N}-N_i)]}$. The highest field of word k this may be in is

$$f_{k,\text{Min}\{j + (\underline{N}-N_i),N_k\}}$$

Therefore, the range of fields that must be searched in target k is

$$f_{k,\text{Max}\{j - (\underline{N}-N_k),1\}} \quad \text{to} \quad f_{k,\text{Min}\{j + (\underline{N}-N_i),N_k\}}$$

An associative search occurs in the same field in all words (unless masked). Because of this the first field searched must be the smallest for all words, and the last the largest. Let

$$N_{min} = \text{Min} \{ N_k \}$$

$$N_{max} = \text{Max} \{ N_k \}$$

The range is now (field numbers)

$$\text{Max}\{j - (\underline{N}-N_{min}),1\} \quad \text{to} \quad \text{Min}\{j + (\underline{N}-N_i),N_{max}\}$$

or,

$$p_{ij} = \text{Min}\{j + (\underline{N}-N_i), N_{max}\} - \text{Max}\{j - (\underline{N}-N_{min}), 1\} + 1 \qquad (6\text{-}1)$$

parallel associative searches must be done to insure that all fields that could contain a value matching $e_{ij}$ have been checked. Checking all fields in word i requires

$$p_i = \sum_{j=1}^{N_i} [\text{Min}\{j+(\underline{N}-N_i), N_{max}\} - \text{Max}\{j-(\underline{N}-N_{min}), 1\}] + N_i \qquad (6\text{-}2)$$

searches. Doing this for all words gives

$$p = \sum_{i=1}^{NCT} \left\{ \sum_{j=1}^{N_i} [\text{Min}\{j+(\underline{N}-N_i), N_{max}\} - \text{Max}\{j-(\underline{N}-N_{min}), 1\}] + N_i \right\} \qquad (6\text{-}3)$$

total parallel searches.

There is an inconsistency with the above equation in that it assumes that all indices in all targets will eventually be employed as the comparand in a search. This obviously allows a multiple number of searches for the same index value. In practice, once a measurement index (and its associated targets) is added to a cluster it should be deleted from all target lists. This is done by using the search response register to create a mask to clear the appropriate fields. The effect of deleting indices is to shorten the index lists of the targets. Therefore, the value of $N_i$ should be interpreted as the remaining number of indices in the list for target i, given the processing for all previous targets in the specific cluster. It should also be noted that this number, as well as the total number of searches, is highly dependent on the ordering of the targets within the AM.

As Eq. 6-2 is inspected closely, it becomes evident that there is only a substantial savings in searches when $(\underline{N}-N_i)$ and $(\underline{N}-N_{min})$ are small. This occurs when each list contains almost all possible elements, thereby narrowing down the set of values that could have been in the field and the set of fields in which to search. The limiting case arises when all lists contain exactly $\underline{N}$ indices. We may generalize this to conclude that as the number of reports per target increases (keeping the number of reports fixed) the number of required searches decreases. And as the number of reports per target increases, the number of clusters will decrease. That is, the targets have more common reports. Therefore, the greatest savings in searches occurs for scenarios in which clustering itself provides the least benefit.

A second method to reduce the number of required searches is to employ the value of comparand to determine the feasible fields in which to search. If we let

$$e_{ij} = v_\ell$$

be the value of the index found in $f_{ij}$, then the fields to be searched are now

$$\text{Max}\{v_\ell - (\underline{N}-N_{min}),1\} \quad \text{to} \quad \text{Min}\{v_\ell, N_{max}\}$$

and so,

$$p = \sum_{\ell=1}^{\underline{N}} [\text{Min}\{v_\ell, N_{max}\} - \text{Max}\{v_\ell-(\underline{N}-N_{min}),1\}] + \underline{N} \qquad (6\text{-}4)$$

parallel searches are required. This value will be less than that of Eq. 6-3 as the uncertainty in the index value has been removed. The value is also independent of the target ordering. For scenarios in which there are many reports associated, but with few associated with any one target, Eqs. 6-3 and

6-4 both reduce to searching all measurement index fields. Such scenarios either have many clusters or loosely connected targets within the clusters. In such cases it appears advantageous to simply search all fields without attempting to determine the best search strategy, and so avoid the computational penalty. Obviously, if all fields are to be examined,

$$p = \underline{N}*N_{max} \qquad (6\text{-}5)$$

searches are required.

We have described the search portion of the index clustering method, but have up to this point omitted the overall processing scheme. The following is a step by step breakdown of the suggested implementation, assuming the target data structure has already been created (c.f., Fig. 6-3). Any of the three search strategies mentioned may be employed.

1.0 Clear the M, R, T, and Comparand Mask registers.

Initialize a cluster counter (register) to 1.

Initialize a current target indicator (register) to 1.

2.0 Loop over all index fields in the current target.

2.1 If the current field is empty, go to 2.0.

2.2 Load the index in the field into the Comparand.

2.3 Determine the fields over which to search, using any of the methods given,

2.4 Loop over feasible fields.

2.4.1 Perform associative EQUAL search on the current field for all targets.

2.4.2 Logical OR the R register with the T register.

2.4.3 Copy the R register into the M register, invert, and use to clear the current field in all words (including the current one).

3.0 Insert the current cluster number into the current target's field. Clear the bit flag in the T register corresponding to this word.

4.0 If there are any bits set in the T register, then set the target indicator to the target number corresponding to the first bit set in the T register. Go to 2.0.

5.0 If there are no bits set in the T register, then the current cluster is complete. Using the values stored in the Cluster number field, determine the unclustered targets. If there are any still unclustered, set the target counter to the first available target. Go to 2.0.

6.0 If there are no remaining unclustered targets, stop.

A worst case of $\underline{N}*N_{max}$ parallel EQUALS searches are necessary for the index clustering, while the bit flag clustering method has a worst case of NCT ANY searches. Based solely on the number of searches, the bit flag method is superior. The number of Confirmed targets will be much less than the number of returns in the NSCAN + 1 scans of interest.

POSITION CLUSTERING

Both bit flag clustering and index clustering suffer from fairly large data structures: bit flag clustering can involve substantial flag vectors in both the track and target data structures, while index clustering can have large index lists within the target data structure. An alternate approach to clustering will be suggested here in an attempt to alleviate these storage problems. We will not study this particular approach in great detail, and present it mostly as an example.

The purpose of clustering is to divide the global hypothesis formation function into smaller, independent functions. The precise solution to this problem is to create clusters with disjoint sets of associated sensor returns. A more relaxed solution is to form clusters of targets such that any two targets in separate clusters have a low probability of having a common associated

return. This view is motivated by a standard illustration of clustering in which it is stated that targets in grossly different sections of the environment need not be considered together in forming global hypotheses. Therefore, if targets can be grouped in distinct regions in space, then the global hypotheses can be formed independently within these groups. It could be that these groups contain common returns, and so the overall global hypothesis formation is done under incorrect assumptions, but the effect of this will be small. Note that we are not stating that the return indices (or bit flags) can be totally eliminated; they are still necessary for global hypothesis formation. What we are eliminating is the associated return lists within the target data structure needed in the first two methods.

The position clustering algorithm partitions the x-y parameter space into disjoint spatial regions. Targets in separate regions will have a low probability of containing a common associated return. As we have stated, it will be possible for targets in different regions to have a common report. It will also be possible that a target may be completely independent from all other targets in the cluster. These inaccuracies are tolerated in order to reduce the target word size. Note that we will only consider the partitioning of the x-y space. This approach can be extended to include all state parameters (e.g., height, velocities, id, etc.), though we will concentrate on the more intuitive planer positions.

The first step to creating these connected regions is to create a region for each target tree. Target trees themselves have no states with which to work; state information is associated with the tracks within the tree. But clustering must occur over targets and not tracks, and so aggregate target regions must be formed. This could be done by employing the state estimate

for each track and determining the region that includes all the track estimates for each target. Such a method ignores the uncertainties in the track estimates. A more attractive approach would be to determine the target region that includes all track error ellipses of a preset size (e.g., 3 sigma). This resulting target region is then simply be the union of these individual track ellipses. Unfortunately, a collection of ellipses is too involved to reasonably manage. Therefore, the target region will be defined to be the smallest rectangle that encloses the track error ellipses. Each target region can then be characterized by four parameters--the minimum and maximum values for x and y. By employing rectangles we will simplify the computations at the expense of enclosing more of the x-y space than is truly necessary.

Figure 6-4 is an example of these rectangular target regions for two targets. The shaded regions represent the track individual uncertainties. Ellipses are not strictly required for track uncertainties as we will only employ the extreme x and y values on the ellipses, and so rectangular track regions may be used.

Once the target regions are formed, all intersecting regions should be combined into clusters. Again, we will force the resulting regions to be rectangular. As shown in Fig. 6-4, these cluster regions (demarcated by the dotted lines) contain more space than in the original target regions. A consequence of this excess space could be that targets that do not actually intersect are combined. This will not create any errors per se, but will cause targets to be processed in the same cluster that need not be processed together, increasing the computation time.

Table 6-3 contains the fields necessary for the target data structure.

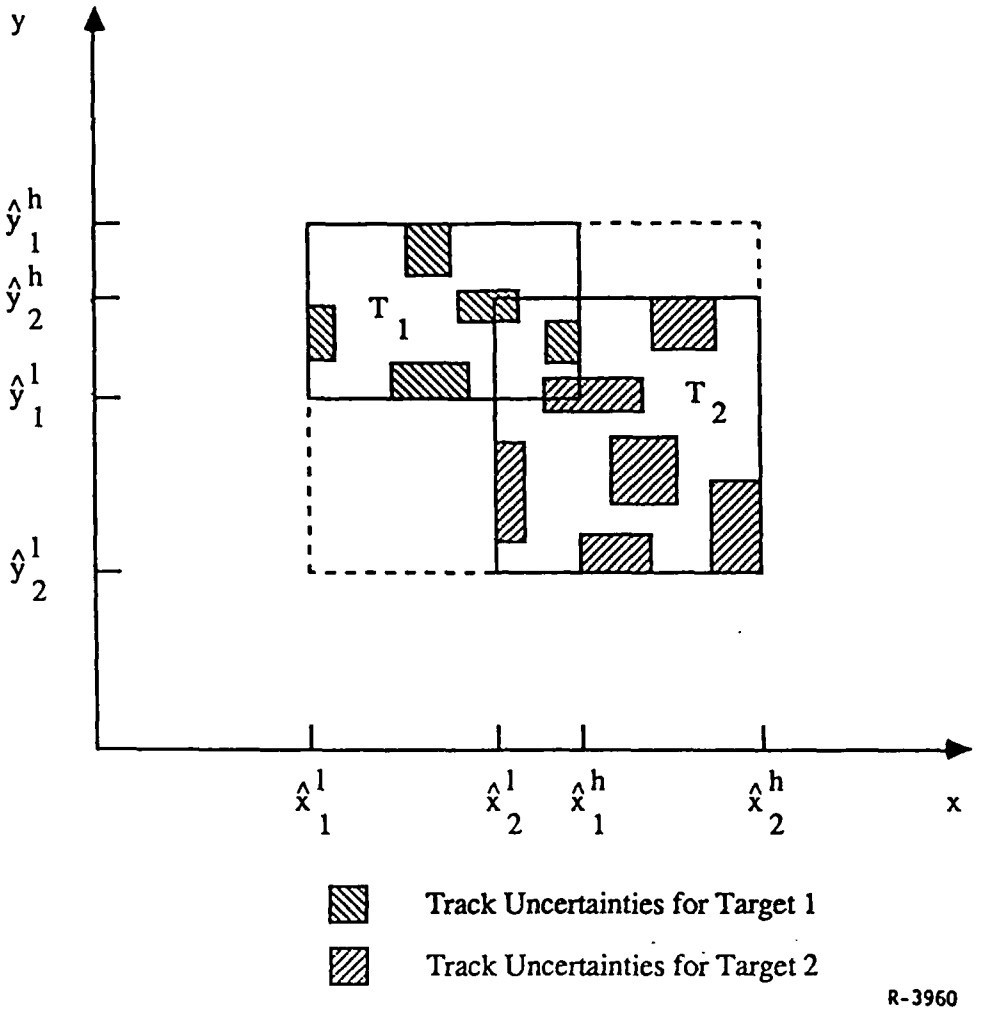



Figure 6-4. Position Clustering Example

TABLE 6-3. TARGET DATA STRUCTURE FOR POSITION CLUSTERING

| Field | Type | Number Required |
|---|---|---|
| Target Number | Integer | 1 |
| Cluster Number | Integer | 1 |
| Region Boundaries | Real | 4 |

To create the target data structure, it is first necessary to form the individual track regions. For each track calculate ,in parallel

$$\hat{x}^{l} = \hat{x} - \gamma\sigma^{xx}$$

$$\hat{x}^{h} = \hat{x} + \gamma\sigma^{xx}$$

$$\hat{s}^{l} = \hat{y} - \gamma\sigma^{yy}$$

$$\hat{y}^{h} = \hat{y} + \gamma\sigma^{yy}$$

where

$\gamma$ = "size" of uncertainty region

$\sigma^{xx}$ = variance of x estimate

$\sigma^{yy}$ = variance of y estimate

This calculation will require

| | |
|---|---|
| 4 | multiplies |
| 4 | additions |

The results should be stored in four temporary fields in the track words.

Once the individual track regions are formed they must be combined. This is efficiently handled in the AP by searching for the maximum values in the $\hat{x}^{h}$ and $\hat{y}^{h}$ fields, and the minimum values in the $\hat{x}^{l}$ and $\hat{y}^{l}$ fields, for each target in turn. These results are the boundaries of the target regions, and so should be stored in the target data structure.

The next step in position clustering is to form the interconnected cluster regions. We will employ a method functionally similar to the bit flag clustering algorithm, but instead of searching for set intersections, we are

searching for region intersections. On each iteration of the algorithm, the first available unclustered target is employed, and all intersecting targets with this region are found. These individual regions are then combined, and the resulting boundaries stored in the target structure, overwriting the original target boundaries. The targets combined are then masked out of all subsequent searches. This new region is now employed again to see if it has any intersecting targets. If there are, the processing continues as before. If not, the processing for this cluster terminates and the current cluster number is inserted into the appropriate cluster field.

### 6.3.5 Global Hypothesis Formation

Clustering of targets allows the formation of global hypotheses to be divided into independent sections. Unfortunately, the SIMD nature of APs forces clusters to be processed sequentially, not concurrently. Clustering is useful, though, as we have replaced a large combinatorial problem with several smaller ones. In section we will discuss the global hypothesis formation for a single cluster.

The formation of global hypotheses for a single cluster is basically an exhaustive search for all disjoint track combinations. Disjoint is defined to mean no two tracks in a candidate hypothesis are from the same target tree or include a common return. The search capability of the AM can determine the allowable additions to a partial hypothesis in two steps - one to eliminate all the tracks belonging to the same target tree as the last track added to the hypothesis, and one to eliminate all tracks having common measurements with the last track. This assumes that a mask (running list) of allowable tracks is updated every time a new track is added, and stored for future use.

Global hypotheses are equivalent to directed graphs, where each level of the graph corresponds to a target and the nodes at that level are the track nodes of that target. The allowable paths are determined by the previous nodes (i.e., the associated track measurements) on the path. The value placed on visiting a node is the negative log of the likelihood. We employ the logs of the likelihoods so that the overall likelihood can be accumulated by addition instead of multiplication, and the negative of the log so that the global hypothesis function becomes a minimization problem.

Figure 6-5 is an example of one hypothesis in such a graph. The existence of an edge between any two nodes means that the path is allowable based on the previously visited nodes. The source and sink nodes are inserted merely to complete the graph, and have no value.

A possible global hypothesis generation scheme will be presented here. The bit flag method of storing associated returns will be employed. A depth first search will be used, which will require that each level (target) have associated with it a mask of allowable descendant tracks based upon disjoint measurements and nodes that have already been traversed, and a partial hypothesis score (likelihood). In this way the track masks need not be reconstructed from scratch whenever a new path is considered, but instead can be computed in one step from the mask at the level above and the new track. A track mask and partial likelihood will require

| | |
|---|---|
| $NCT*B_c$ | Bits |
| 1 | Real |

for each target, where Bc is the number of branches per Confirmed target. Actually the above overstates the memory requirements as target i only needs

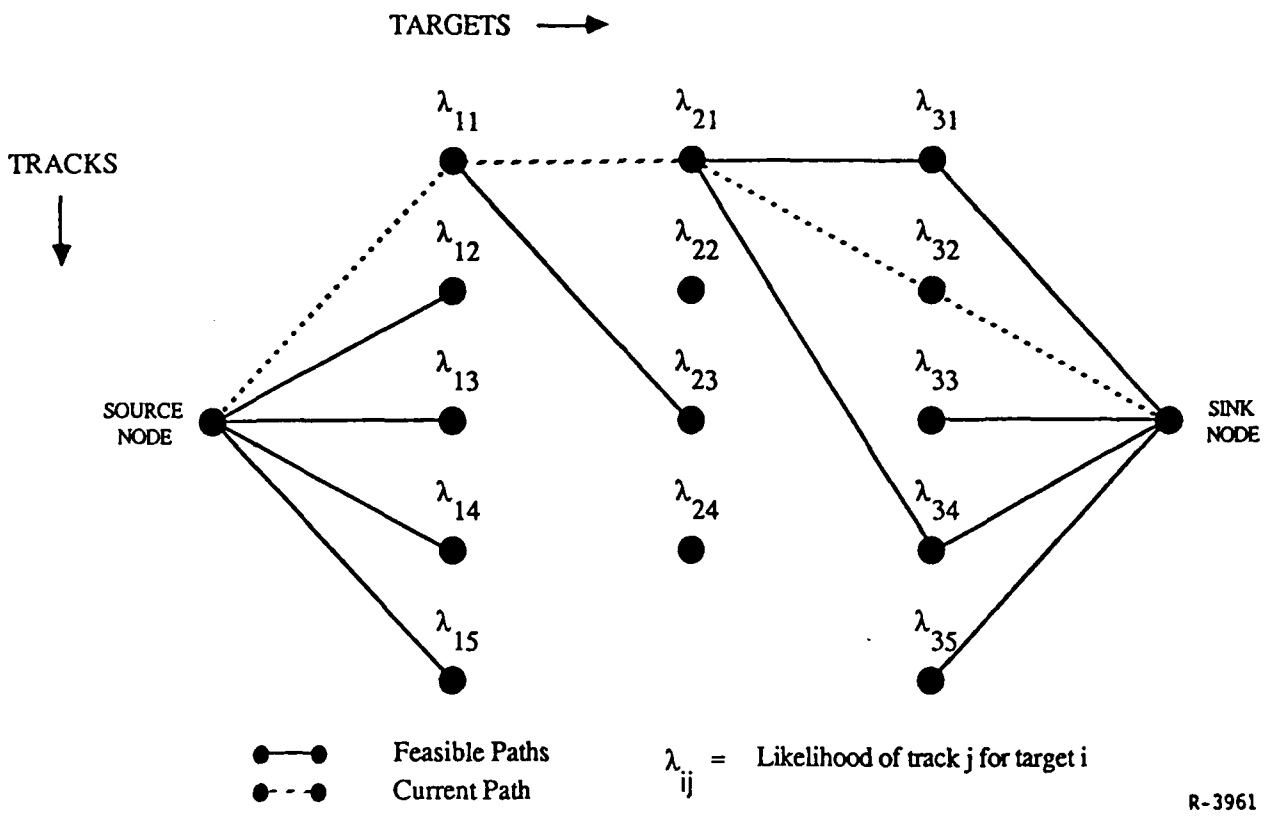



Figure 6-5. Graph Analogy for Global Hypotheses

the tracks for target i + 1 through NCT (i.e., its descendents). Additionally, the current best hypothesis must be stored. This will consist of the hypothesis likelihood and NTC track numbers.

The first step in the algorithm is create the first candidate hypothesis. The return bit flags for the first track of the first target are used to determine all feasible (disjoint) tracks that can be combined with itself. This track mask, along with the likelihood of this track, are stored within the target data structure. Next, the first feasible track of the second target is activated. The bit flags for this track are loaded into the Comparand, and the allowable tracks for this track alone are found. By forming the

logical AND of this mask and the one for the target preceding it, the disjoint tracks for the pair may be found. This mask, and the sum of the previous partial likelihood and the current one, are stored in the present target's data structure. This processing continues until a track from the last target has been added. Once this last track is added, a full hypothesis is complete and becomes the best hypothesis found so far.

After the first hypothesis is evaluated the last track added is deleted from the candidate hypothesis and the next allowable track from the same target is considered. The best (highest combined likelihood) global hypothesis found is stored and compared to subsequent hypotheses. This continues until no more allowable tracks exist for that target, when the algorithm backtracks to the previous target's tracks. When all legal track combinations have been formed, the process stops. Bounding techniques should be employed in practice to limit the breadth of the graph and the depth of some searches.

Two associative searches are required at each step of the global hypothesis process: one to activate the next target to be added (associative EQUALS) and one to determine the feasible additions based on measurements (associative logical ANY). Once these searches are performed they do not have to be repeated at that target level until all completions of the current partial hypothesis have been formed. The formation of global hypotheses requires, at most,

$$\sum_{i=1}^{NTC-1} B_c = B_c*(N_c - 1) \qquad (6-6)$$

searches of each type. This bound is loose, as the underlying assumption is that all possible track combinations from different targets are allowable. But if this were true the cluster formation process would not have combined the target trees. A better approximation is to assume that the addition of a track to an existing partial hypothesis reduces the allowable tracks per target by an average fraction f. Therefore, the required number of searches of each type is now

$$\sum_{i=1}^{NTC-1} B_c*(1-f)^{i-1} = B_c* \left[ \frac{1 - (1-f)^{NTC-1}}{f} \right] \qquad (6-7)$$

The value of f is highly dependent on the scenario. For instance, targets flying in formation create many tracks with conflicting measurement assignments, causing f to be quite large.

In addition to the searches, the individual likelihoods must be combined to form global hypotheses. This will take

$$\sum_{i=1}^{NTC} B_c*(1-f)^{i-1} = B_c* \left[ \frac{1 - (1-f)^{NTC}}{f} \right] \qquad (6-8)$$

additions assuming the partial hypothesis likelihoods are stored.

This global hypothesis procedure is similar to that proposed in [52] for 0-1 integer programming on APs.

### 6.3.6 Pruning

Confirmed target tracks not included in the most likely global hypothesis NSCANs in the past are pruned away. Again, the AP handles this easily by performing searches on the measurement flags corresponding to the oldest scan (the more recent scans can be masked out) for each confirmed target. If a

track contains a conflicting return its active track flag is set to 0, making that word available.

### 6.3.7 Track Promotion

The promotion of targets to higher classifications is merely a matter of incrementing the age of all tracks by 1. Also, the measurement flags for the tracks must be shifted over by one scan to eliminate the oldest group and make room for the next scan.

## 6.4 CONCLUDING REMARKS

The intrinsic parallel structure of the track-oriented multiobject tracking algorithm makes it quite applicable to implementation on associative processors. In this section we have given possible mappings of the tracking algorithm onto the specific computer architecture of associative processors. The choice of APs over other computer designs was based upon their past and present use in tracking systems. While these other applications employed much simpler algorithms, the AP architecture has been shown to handle the track-oriented tracking approach quite well. The primary concern with APs is in the clustering and global hypothesis formation functions, as these are the most sequential portions of the algorithm. We have presented several methods of structuring the data and computations in order to exploit the capabilities of the AP. In the next section we will consider the more general architectures of MIMD computers and how the tracking algorithm may be implemented on such systems.

SECTION 7

# ADAPTATION OF THE TRACK-ORIENTED MULTIOBJECT TRACKING ALGORITHM TO MIMD COMPUTERS

## 7.1 INTRODUCTION

In the previous section we discussed possible implementation methods for the track-oriented multiobject tracking algorithm with regard to associative processors. The choice of associative processors was made based upon their historical application to the tracking problem. While it was shown that APs can be employed efficaciously, we do not contend that they are the optimal choice in computer hardware. In fact, it is impossible to select a single best architecture. While some architectures may appear more suited to the tracking problem than others, various factors must be considered in the decision. Considerations such as cost and relative speed must be appraised. For instance, it may be that the high end pipelined computers (e.g., Crays) will out perform most multiprocessors, even though the multiprocessor seems better matched to the problem at hand. Such factors are out of the scope of this research.

In this section we will examine how the track-oriented tracking algorithm can be restructured to exploit the capabilities of MIMD computers. The computer model we will consider is one of a RAM MIMD machine wherein all processing elements can access the required data without conflict. This can be accomplished either with shared memory or global memory configurations with parallel read access, or by local PE memories that contain the required

data. As was done in the previous section, we will ignore the time required for data manipulation and concentrate on the arithmetic execution requirements. Also, in keeping with one of the basic assumptions of this research, we will only consider functional, or large grain, parallelism here. This model is extremely relaxed, without major restrictions in memory, processing, or interconnection networks. In an initial study such as this it is desirable to examine parallelism limited only by the application itself. Such work can then lay the foundation for more realistic hardware restrictions.

In the remainder of this section we will present restructurings of the tracking algorithm that are suited to MIMD computers. Possible problems and concerns germane to MIMD applications will also be presented. The majority of the discussion will concern the functions of clustering and global hypothesis formation. As has been obvious in Sections 5 and 6, the track functions of predicting, gating, and updating can be handled concurrently by target track. It is the synchronization tasks of clustering and global hypothesis formation that present the greatest challenge.

## 7.2 TRACK-LEVEL FUNCTIONS

The term "track-level functions" is defined to be those functional tasks that display a natural parallelism at the track level. As was shown in Fig. 5-2 and discussed in subsection 5.2, these functions are those of predicting the track ahead in time, gating the returns for the current scan, and updating the target tracks with the associated returns. In this subsection we investigate the application of MIMD computers to these functions.

Prediction of the track information is an independent computation for each track, assuming that the necessary state transition and noise matrices

are available to each process. As such values are typically "hard-wired" into the code itself there are no access contention concerns. The only data required is the individual track state estimate vector and covariance matrix. The result of the prediction function is new state estimates and covariances, which may be inserted into the same memory locations that the input data occupied. Therefore, there are no write contention problems.

The total time to predict all tracks is given by

$$T_p = t_p^* \left\lceil \frac{N_t}{N_{pe}} \right\rceil \qquad (7\text{-}1)$$

where

$t_p$ = Average time to predict one track forward in time

$N_t$ = total number of tracks to be predicted

= $N_c^*B_c + N_i^*B_i + N_t^*B_t$

$N_{pe}$ = Number of available PEs

Equation 7-1 takes a form much like a step-function, wherein discontinuous increases in execution time occur at multiples of $N_t$. An increase of one track above $N_t$ will result in an increase in execution time of $t_p$. This is a standard characteristic of multi-processor architectures, and points out the need to match computer and problem size.

The next track-level function is that of gating sensor returns against the predicted track values. Gating differs from prediction in that the computations are not a function of the track data alone. In fact, the total possible parallelism in gating is determined by the total number of tracks and the

total number of returns. Ignoring for the moment problems of write contention, the total time to gate all returns against all tracks is

$$T_g = t_g * \left\lceil \frac{N_t * NR}{N_{pe}} \right\rceil \qquad (7\text{-}2)$$

where

$t_g$ = Average time to gate one return against one track

The processing assumed in Eq. 7-2 is that each processor will perform all the required gating tasks for one track-return pair. These functions include both setting up the gate bounds and performing the comparison with the return. Under this assumption (and given that all returns in the scan are of the same accuracy) the identical gate set-up computations for a single track will be repeated NR times by NR processing elements. Another method would be for a single PE to compute the gate bounds and communicate the results to NR - 1 other processors, or store the results in memory to be accessed by the other PEs when needed. Such a method requires

$$T_g = t_{su} * \left\lceil \frac{N_t}{N_{pe}} \right\rceil + t_c * \left\lceil \frac{N_t * NR}{N_{pe}} \right\rceil \qquad (7\text{-}3)$$

where

$t_{su}$ = Average time to set-up gate bounds for one track

$t_c$ = Average time to compare one return against one set of track bounds

Note that

$$t_g = t_{su} + t_c \qquad (7\text{-}4)$$

And so, given a sufficient number of PEs, Eqs. 7-2 and 7-3 will produce the same execution time. But in the event that

$$N_{pe} < N_t * NR$$

the second method will require less computation time as the common gate set-up calculations are done only once.

A possible concern in the gating function is that, implicitly, it is in this step that the target trees are expanded. If there are enough processors to assign one track-return pair to each PE, the problem of how to determine the labels of the resultant branches arises. Recall that new target tracks are only created for returns that pass the gating tests (we are ignoring missed detection branches as they are an insignificant addition to processing requirements). In conventional computer applications of the track-oriented algorithm, new tracks are simply numbered sequentially as they are created. In MIMD computers, it is not feasible for each PE to independently assign a track label to its track(s), as the PE is not aware of the results of other PEs. It may be possible for new track labels to be assigned based upon the existing track label and the return number. This will undesirably allow for track labels on empty tracks since not all track-return pairs are associated. Alternatively, all resultant new tracks can be communicated to a central PE that inserts them into memory locations.

It is important to note that this is not truly a write contention issue. Write contention refers to two or more PEs attempting to simultaneously update the same variable in the same memory location. Here the problem is one of bookkeeping to avoid writing different variables to the same memory address.

The final track-level function is updating the individual tracks with the appropriate return values. Updating is functionally similar to prediction in that the track estimates and covariance are changed and then allowed to overwrite the original values in memory. The update calculations depend upon the individual track data and return value. The total time necessary to complete updating of all tracks is

$$T_u = t_u^* \left\lceil \frac{N_t}{N_{pe}} \right\rceil \qquad (7\text{-}5)$$

where

$t_u$ = Average time to update one track

## 7.3 CLUSTER FORMATION

The function of cluster formation differs from the track-level functions in that it is not parallel by track. In fact, the algorithm currently employed in sequential computers displays little if any functional parallelism. In this subsection we will discuss possible restructurings of the clustering algorithm that promote MIMD computer applications.

Clusters can be formed most economically by first forming subclusters during the association (gating) stage. Subclusters are basically clusters formed on the basis of one scan of data. For each return in a scan all targets which accept the return for update are grouped together into a single subcluster. Since different tracks in a single target tree may be updated with different returns, the same target will appear in several subclusters. A target may also be included in subclusters in each of the NSCAN + 1 scans of interest.

The purpose of clustering is to combine all subclusters that have common targets. This is in contrast to the definition of clustering used in the AP applications of the previous section, where the content addressing capability of AMs made clustering directly from the associated returns more attractive.

Clustering may be viewed as constructing an interconnection matrix, M, where its elements are given by

$$m_{ij} = \begin{cases} 1 & \text{if } C_i \cap C_j \neq 0 \\ 0 & \text{else} \end{cases} \qquad (7\text{-}6)$$

and

$C_i$ = Subcluster i (set of targets)

$1 \leq i \leq NS$

M is obviously symmetric with 1's on the diagonal, so we therefore need only consider the region above the diagonal. To determine the interconnections it is necessary to compare each subcluster $C_i$ with all other subclusters $C_j$ where $j > i$, thereby finding each row in the upper triangular section. This requires

$$\sum_{i=1}^{NS} (NS - i)$$

set compares. Note that

$$\sum_{i=1}^{NS} (NS - i) = \binom{NS}{2} = \frac{NS(NS - 1)}{2}$$

which, given a worst case of NTS compares per set (NTS is the number of Confirmed targets per subcluster) is the same as the operation count in subsection 5.3. Also, as in subsection 5.3, we will ignore the actual merging of the subclusters and concentrate solely on determining the subclusters to be merged.

Computing each full row of the interconnection matrix determines all subclusters that have a common target with the subcluster corresponding to that row. Computing a single element of the matrix determines whether those two subclusters are interconnected. Given both sufficient data availability and sufficient processors, each element of the matrix may be computed independently from all others. The results may then be combined to form M.

If the smallest divisible unit of computation is the determination of a single element in the interconnection matrix, then the total number of operations that are required is at most

$$NTS \left\lceil \frac{NS(NS - 1)}{2N_{pe}} \right\rceil$$

16 bit comparisons. Equivalently, the time required to complete the relevant section of M is

$$T_c = t_c \left\lceil \frac{NS(NS - 1)}{2N_{pe}} \right\rceil \qquad (7\text{-}7)$$

where

$t_c$ = Average time to compare two different subclusters

An important characteristic of MIMD computers that has not been explicitly mentioned is their asynchronous nature. This capability will prove very important in clustering, as the subcluster set comparison operation halts

whenever a match is found. Therefore, some operations will stop after one comparison while others will go through all targets in the sets. Because of the asynchronous control of the processing new tasks (e.g., another set compare) can initiate immediately upon completion of the previous process without having to wait for the other PEs to finish.

## 7.4 GLOBAL HYPOTHESIS FORMATION

Global hypothesis formation is the selection of target tracks, one from each target tree in the cluster, that are disjoint (no common reports) and satisfy some optimization criteria. The optimization criteria we employ is that of minimizing the negative log likelihood of the composite hypothesis. In the current sequential implementation the basic algorithmic structure is to form all disjoint sets of tracks and compare the scores. Actually, this procedure is abbreviated somewhat by employing branch and bound techniques, though we will not discuss them here.

The first observation about forming global hypotheses is that each cluster may be processed independently. Doing so gives a required time to complete of

$$T_{gh} = t_{gh}^* \left\lceil \frac{NC}{N_{pe}/N_{gh}} \right\rceil \qquad (7\text{-}8)$$

where

$t_{gh}$ = Average time to create global hypotheses for all clusters

NC = Number of clusters

$N_{gh}$ = Number of PEs required to process one cluster

We make the distinction between the number of available processors, $N_{pe}$, and the number of processors required for global hypothesis formation, $N_{gh}$, because generally more than one PE will be employed in a single cluster.

The question now is how to structure the global hypothesis operations to exploit the MIMD capabilities. In the previous section we introduced a graph model for global hypothesis formation where the connectivity of the graph was dependent on the current active path (c.f. Fig. 6-5). Using this analogy our task becomes one of partitioning the graph such that independent searches may be carried out. Simply subdividing the graph by targets or tracks will not accomplish this as global hypotheses will in general cross over these boundaries. That is, a global hypothesis will contain branches that appear in separate subdivisions of the graph. Due to the path dependent nature of the connectivity it is not possible to simply combine the best partial hypotheses found in each subdivision, and so all partial hypotheses from each partition would have to be stored. After the processing in each subdivision is completed the results could conceivably be combined by testing each partial hypothesis for disjointness. Such a method is considered inefficient and will require a large amount of memory space to store the partial results.

A better approach is to realize that no matter how the computations are distributed, there will have to be some sort of synchronization between processes in order to determine the optimal hypothesis. With this in mind, we look to create optimal <u>conditionally</u> independent global hypotheses and then, after all such best hypotheses have been formed, select the minimum. Conditional independence is most simply illustrated by an example. Let the number of processors equal the number of branches on Confirmed target #1. We may now divide the global hypothesis formation problem into $N_{pe}$ ( = $B_c$) subproblems,

where subproblem i is to find the best hypothesis given that it must include branch i of target #1. Since the optimal solution must contain a branch of target #1, one of the $N_{pe}$ partial solutions must be the optimal one.

This approach is not constrained to use the first Confirmed target in the cluster; any target may be employed. If the number of PEs is greater than the number of branches than several variations may be considered. The first is to simply ignore the extra PEs. This gives a total time requirement of

$$T_{gh} = \frac{T_{gh}}{B_c} * \left\lceil \frac{B_c}{N_{pe}} \right\rceil \qquad (7\text{-}9)$$

There is an implicit assumption in the above equation that the individual conditional hypotheses all require the same amount of time to compute. This may not be the case, as the processing time is a function of the number of possible conditional hypotheses in each PE, which will obviously vary. Nevertheless, the average time is considered an acceptable estimate.

A second method of employing processing elements is to form more than $B_c$ conditionally independent subproblems. This is a more difficult approach than the first. The number of conditionally independent subproblems based on the tracks of one Confirmed target is $B_c$. If the tracks from two targets are combined then

$$(B_c)^2(1 - f)$$

subproblems may be formed, where the approximation is made that the addition of a track to an existing partial hypothesis reduces the allowable tracks per target by an average fraction f. This will produce a total required time of

$$T_{gh} = \left\{ \frac{T_{gh} - t_{su}}{(B_c)^L(1-f)^{L-1}} \right\} \left\lceil \frac{(B_c)^L(1-f)^{L-1}}{N_{pe}} \right\rceil + t_{su}, \quad N_{pe} > B_c \qquad (7\text{-}10)$$

where

$$L = \left\lceil \frac{\lg(N_{pe}) + \lg(1-f)}{\lg(B_c) + \lg(1-f)} \right\rceil , \quad N_{pe} > B_c \qquad (7\text{-}11)$$

$t_{su}$ = Average time to set up subproblems

For $N_{pe} < B_c$, Eq. 7-9 holds. L is the number of target branches upon which the partial hypotheses are conditioned. Note that for $N_{pe} = B_c$, Eqs. 7-9 and 7-10 are equivalent. A major concern with creating more than $B_c$ conditionally independent subproblems is that an increasingly complex set-up must be accomplished first. In the extreme case with an unlimited number of PEs, each disjoint candidate hypothesis must be identified before the subproblems may be created. But if this is done prior to the parallel portion all that remains for the PEs to calculate is the likelihood sums. That is, practically all the global hypothesis formation has been done sequentially in set-up, and so $t_{su}$ will approach $T_{gh}$. This set-up time may be avoided at the cost of wasting processing power by not determining the disjoint candidate hypotheses ahead of time, but just assign candidate hypotheses to PEs whether they are disjoint or not. The individual PEs must then test to ensure that the basis of their conditional hypothesis is sound. If the basis tracks for a particular hypothesis are not disjoint, processing will halt and the PE will sit idle.

A final method of handling the subproblems with $N_{pe} > B_c$ depends upon the ability of some MIMD architectures to dynamically allocate tasks to the processors. Such a configuration would alleviate the problem of PEs sitting idle mentioned above. Whenever a PE completes its assigned task it reports that it

has finished to its controlling process, which may then assign it a new task. This suggests a hierarchical arrangement of processors. The higher level processors will take whatever subproblem is assigned to them, and divide it further for implementation on their slaved processors. Unfortunately, such asynchronous control is difficult to accurately investigate at the level of generality we are interested in here.

## 7.5 CONCLUDING REMARKS

The adaptation of the track-oriented multiobject tracking algorithm to MIMD computers is a complicated task that, in actual applications, will be heavily driven by the specific architecture to be employed. In this section we have described possible implementation methods and discussed relevant issues that arise in such systems. The analysis provided here will hopefully form the basis of further study into restructurings of the tracking algorithm for fully defined systems. Once such issues as processor capabilities, memory configurations, and communication networks are well understood (as will happen when working on a real, physical computer), the functions studied here may be derived and examined in far greater detail.

REFERENCES

1. Delaney, J.R., A.S. Willsky, A.L. Blitz, and J. Korn, "Tracking Theory for Airborne Surveillance Radars," ALPHATECH Technical Report TR-142, Burlington, Massachusetts, February 1983.

2. Washburn, R.B., T. Kurien, A.L. Blitz, and A.S. Willsky, "Hybrid State Estimation Approach to Multiobject Tracking for Airborne Surveillance Radars," ALPHATECH Technical Report TR-180, Burlington, Massachusetts, October 1984.

3. Kurien, T., A. Blitz, and R. Washburn, "Optimal Maneuver Tracking Using Passive Sensors," Proceedings of the 6th MIT/ONR Workshop on C3 Systems, Cambridge MA, June 25-29, 1983, pp. 164-171.

4. Kurien, T. and R.B. Washburn, "Multiobject Tracking Using Passive Sensors," Proceedings of the 1985 American Control Conference, Boston MA, June 19-21, 1985, pp. 1032-1038.

5. Hwang, K., and F.A. Briggs, Computer Architecture and Parallel Processing, McGraw-Hill Book Company, New York, 1984.

6. Flynn, M.J., "Some Computer Organizations and Their Effectiveness," IEEE Transactions on Computers, Vol. C-21 No. 9, Sept. 1972, pp. 948-960.

7. Zakharov, V., "Parallelism and Array Processing," IEEE Transactions on Computers, Vol. C-33 No. 1, Jan. 1984, pp. 45-78.

8. Potter, J.L., ed., The Massively Parallel Processor, The MIT Press, Cambridge MA, 1985.

9. Kuck, D.J., and R.A. Stokes, "The Burroughs Scientific Processor (BSP)," IEEE Transactions on Computers, Vol. C-31 No. 5, May 1982, pp. 363-376.

10. Siegal, H.J., "Interconnection Networks for SIMD Machines," Computer, Vol. 12 No. 6, June 1979, pp. 57-65.

11. Feng, T., "A Survey of Interconnection Networks," Computer, Vol. 14 No. 12, Dec. 1981, pp. 12-27.

12. Batcher, K.E., "Design of a Massively Parallel Processor," IEEE Transactions on Computers, Vol. C-29 No. 9, Sept. 1980, pp. 836-840.

13. Foster, Caxton C., Content Addressable Parallel Processors, Van Nostrand Reinhold Company, New York, 1976.

14. Yau, S.S. and H.S. Fung, "Associative Processor Architecture--A Survey," Computing Surveys, Vol. 9 No. 2, March 1977, pp. 3-27.

15. Batcher, K.E., "STARAN Parallel Processor System Hardware," Proceedings of the National Computer Conference, V. 43, 1974, pp. 405-410.

16. Batcher, K.E., "The Flip Network in STARAN," Proceeding of the 1976 International Conference on Parallel Processing, Aug. 24-27, 1976, pp. 65-71.

17. Bauer, L.H., "Implementation of Data Manipulating Functions on the STARAN Associative Processor," Proceedings of the Sagamore Computer Conference, Aug. 20-24, 1974, pp. 209-227.

18. Enslow, P.H., "Multiprocessor Organization--A Survey," Computing Surveys, Vol. 9 No. 1, March 1977, pp. 103-129.

19. Crowther, W., et. al., "Performance Measurements on a 128-Node Butterfly Parallel Processor," Proceedings of the 1985 International Conference on Parallel Processing, pp. 531-540.

20. Wu, C. and T. Feng, Tutorial: Interconnection Networks for Parallel and Distributed Processing, IEEE Computer Society Press, Silver Spring, Maryland, 1984.

21. Evensen, A.J., and J.L. Troy, "Introduction to the Architecture of a 288-Element PEPE," Proceedings of the 1973 Sagamore Conference on Parallel Processing, August 22-24, 1973, pp. 162-169.

22. Crane, B.A., et. al., "PEPE Computer Architecture," Proceedings of the Sixth Annual IEEE Computer Society International Conference, San Francisco, CA, Sept. 12-14, 1972, pp. 57-60.

23. Dingeline, J.R., H.G. Martin, and W.M. Patterson, "Operating System and Support Software for PEPE," Proceedings of the 1973 Sagamore Conference on Parallel Processing, Aug 22-24, 1973, pp. 170-178.

24. Wilson, D.E., "The PEPE Support Software System," Proceedings of the Sixth Annual IEEE Computer Society International Conference, San Francisco, CA, Sept. 12-14, 1972, pp. 61-64.

25. Bergland, G.D., and C.F. Hunnicutt, "Application of a Highly Parallel Processor to Radar Data Processing," IEEE Transactions on Aerospace and Electronic Systems, Vol. AES-8 No. 2, March 1972, pp. 161-167.

26. Cornell, J.A., "Parallel Processing of Ballistic Missile Defense Radar Data with PEPE," Proceedings of the Sixth Annual IEEE Computer Society International Conference, San Francisco, CA, September 12-14, 1972, pp. 69-72.

27. Heimerdinger, W.L., et al, "Architectural Considerations in Interfacing a Parallel Processor to the Air Traffic Control System," Proceedings of the 1974 Sagamore Computer Conference, August 20-23, 1974, pp. 372-382.

28. Schmitz, H.G., and C. Huang, "An Efficient Implementation of Conflict Prediction in a Parallel Processor," Proceedings of the 1974 Sagamore Computer Conference, August 20-23, 1974, pp. 383-399.

29. Baer, J., Computer Systems Architecture, Computer Science Press, Rockville, Maryland, 1980.

30. Blakely, C.E., "PEPE Applications to BMD Systems," Proceedings of the 1977 International Conference on Parallel Processing, August 23-26, 1977, pp. 193-198.

31. Davis, E.W., "STARAN Parallel Processor System Software," Proceedings of the National Computer Conference, V 43, 1974, pp. 391-396.

32. Batcher, K.E., "STARAN/RADCAP Hardware Architecture," Proceedings of the 1973 Sagamore Conference on Parallel Processing, August 22-24, 1973, pp. 147-152.

33. Feldman, J.D., and O.A. Reimann, "RADCAP: An Operational Parallel Processing Facility," Proceedings of the 1973 Sagamore Conference on Parallel Processing, August 22-24, 1973, pp. 140-146.

34. Summers, M.W., and D.F. Trad, "The Evolution of a Parallel Active Tracking Program," Proceedings of the 1974 Sagamore Computer Conference, August 20-23, 1974, pp. 238-249.

35. Stanke, Z.C., "Automatic Track Initiation Using the RADCAP STARAN," Proceedings of the 1976 International Conference on Parallel Processing, August 24-27, 1976, pp. 187-188.

36. Prentice, B.W., "Implementation of the AWACS Passive Tracking Algorithms on a Goodyear STARAN," Proceedings of the 1974 Sagamore Computer Conference, August 20-23, 1974, pp. 250-269.

37. Katz, R., "Analysis of the AWACS Passive Tracking Algorithms on the RADCAP STARAN," Proceedings of the 1976 International Conference on Parallel Processing, August 24-27, 1976, pp. 177-186.

38. Eddey, E.E., and W.C. Meilander, "Application of an Associative Processor to Aircraft Tracking," Proceedings of the 1974 Sagamore Computer Conference, August 20-23, 1974, pp. 417-428.

39. Boyd, H.N., "An Associative Processor Architecture for Air Traffic Control," Proceedings of the 1974 Sagamore Computer Conference, August 20-23, 1974, pp. 400-416.

40. Batcher, K.E., "Bit-Serial Processing Systems," IEEE Transactions on Computers, Vol. C-31 No. 5, May 1982, pp. 377-384.

41. DiGiacinto, T., "Airborne Associative Processor," AIAA Computers in Aerospace III Conference, Oct. 26-28, 1981, pp. 202-205.

42. Reed, B., "The ASPRO Parallel Inference Engine (P.I.E.) A Real Time Production Rule System," AIAA/ACM/NASA/IEEE Computers in Aerospace V Conference, Oct. 21-23, 1985, pp. 459-464.

43. Finnila, C.A. and H.H. Love, "The Associative Linear Array Processor," IEEE Transactions on Computers, Vol. C-26 No. 2, Feb. 1977, pp. 112-125.

44. Love, H.H., "Radar Data Processing on the ALAP," Proceedings of the 1976 International Conference on Parallel Processing, Aug 24-27 1976, pp. 161-167.

45. Bar-Shalom, Y., "Tracking Methods in a Multiobject Environment," IEEE Transactions on Automatic Control, Vol. AC-23, August 1978, pp. 618-626.

46. Reid, D.B., "An Algorithm for Tracking Multiple Targets," IEEE Transactions on Automatic Control, Vol. AC-24, December 1978, pp. 843-854.

47. Browne, J.C., "Formulation and Programming of Parallel Computations: A Unified Approach," Proceedings of the 1985 International Conference on Parallel Processing, pp. 624-631.

48. Gajski, D.D., and J.K. Peir, "Essential Issues in Multiprocessor Systems," IEEE Computer, June 1985, pp. 9-26.

49. Heller, D., "A Survey of Parallel Algorithms in Numerical Algebra," SIAM Review, Vol. 20 No. 4, October 1978, pp. 740-777.

50. Bierman, G.J., Factorization Methods for Discrete Sequential Estimation, Academic Press, New York, 1977.

51. Bokhari, S.H., "On the Mapping Problem," IEEE Transactions on Computers, Vol. C-30, No. 3, March 1981, pp. 207-214.

52. Morefield, C.L., "Solution of Multitarget, Multisensor Tracking Problems on Associative Processors," Proceedings of the Fourteenth Annual Allerton Conference on Circuit and System Theory, Monticello, IL, September 29-October 1, 1976, pp. 1074-1083.